

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
Nashville, TN, USA

Additional Key Words and Phrases: Large Language Models, ChatGPT, Prompt Patterns, Prompt Engineering

Abstract

Prompt engineering is becoming a critical skill for software developers by facilitating enhanced interactions with conversational large language models (LLMs), such as ChatGPT, Claude, and Gemini. This emerging discipline focuses on crafting *prompts*, which are instructions that guide LLMs in generating precise outputs, automating tasks, and ensuring adherence to specific qualitative and quantitative standards. Prompts are also a form of natural language programming that tailor the dialogue between users and LLMs, optimizing input, output, and interaction dynamics for many computational tasks, such as developing software, analyzing documents, and/or addressing cyber vulnerabilities.

This paper introduces a comprehensive catalog of prompt engineering techniques—structured as a collection of patterns—aimed at addressing common challenges encountered when integrating LLMs into the software development lifecycle. These prompt patterns serve as an effective means for knowledge transfer, similar to software patterns. In particular, they provide reusable solutions to common problems faced in particular contexts, such as output generation and interaction when conversing with LLMs in the domain of software-reliant systems.

This paper provides three contributions to research on—and the practice of—prompt engineering for applying LLMs to aid users performing computational tasks. First, it establishes a framework for documenting and deploying prompt patterns across various domains, focusing on enhancing LLM utility in software development endeavors. Second, it curates a catalog of prompt patterns, validated through successful application in refining LLM interactions and outputs. Third, it explores the synergistic potential of creating more complex prompts by combining multiple prompt patterns.

1. INTRODUCTION

The impact of large language models (LLMs) on the software development lifecycle. Conversational large language models (LLMs) [Bommasani et al. 2021; Zhou et al. 2023], such as ChatGPT, Claude, and Gemini, are generating immense interest in a wide range of domains. Example applications include answering questions on medical licensing exams [Gilson et al. 2022], generating code in multiple languages and programming paradigms [Buscemi 2023], grading programming assignments in upper-level computer science courses [Grandel et al. 2024], and accelerating certain aspects of the software acquisition process [Robert and Schmidt 2024].

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission. A preliminary version of this paper was presented in a writers' workshop at the 30th Conference on Pattern Languages of Programs (PLoP'23), October 22-25, 2023, Allerton Park, Monticello, Illinois, USA. Copyright 2024 is held by the author(s). HILLSIDE 978-1-941652-19-0

LLMs answer questions posed by users via “prompts” [Liu et al. 2023], which are natural language instructions that “program” a conversational LLM by requesting, customizing, enhancing, and/or refining its capabilities. For example, here’s a simple conversation with ChatGPT that demonstrates a prompt followed by its response:

Prompt: “Please briefly explain what a ‘prompt’ is in the context of an LLM like ChatGPT.”

Response: In the context of an LLM like ChatGPT, a “prompt” refers to the input text provided by a user to the model. This input text serves as a request or instruction, guiding the LLM on what kind of information, response, or output the user is seeking. The prompt can range from a simple question, a statement requiring elaboration, a command to generate text in a specific format, or even a complex scenario for the LLM to interpret and respond to.

The LLM processes the prompt, leveraging its pre-trained knowledge and patterns learned from vast amounts of text data, to generate a response that aligns with the user’s request. The quality and relevance of the LLM’s response can significantly depend on how the prompt is structured, including its clarity, specificity, and the context it provides. Prompts are fundamental in interacting with LLMs, acting as the primary interface through which users communicate their queries and tasks to the model.

Prompts not only respond to user questions but also shape future interactions and the outputs produced by LLMs. They guide the direction of the subsequent dialogue and determine the nature of LLM-generated content by establishing clear rules and guidelines from the outset of a conversation. Moreover, prompts establish the conversation’s context, indicating to LLMs which information is crucial and defining the preferred formats and substance of both inputs and outputs.

For example, a prompt could instruct an LLM to only generate code that follows certain coding principles (such as applying the SOLID principles [Martin 2023]) or programming paradigms (such as object-oriented, functional, or reactive programming). Likewise, a prompt could instruct an LLM to flag certain phrases in software requirement documents and provide additional information (such as inconsistencies in definitions and usages) related to those phrases. By guiding interactions with an LLM, prompts facilitate more structured and nuanced outputs to aid a large variety of computational tasks via natural language programming.

Emerging trends, challenges, and opportunities for LLMs. Although conversational LLMs have been generally available only since the end of 2022, they are now widely applied to generate and assess computer programs. For example, LLMs have been integrated into software tools, such as Microsoft/GitHub Copilot [git 2024; Asare et al. 2022; Pearce et al. 2022] and included in integrated development environments (IDEs), such as IntelliJ [Krochmalski 2014] and Visual Studio Code. Software teams can thus access these AI-assisted tools directly from their preferred integrated development environments (IDEs), underscoring the transformative role LLMs now play in enhancing productivity and innovation across the software development lifecycle, from initial coding to testing to final deployment.

Despite the rapid integration of LLMs into software development, a comprehensive understanding of these models—particularly their capabilities and limitations—is still emerging. Viewing LLMs as a computing platform introduces a unique programming paradigm based on natural language, presenting both challenges and learning curves. This early adoption stage highlights the need for rigorous exploration and disciplined knowledge formation so programmers can apply LLMs effectively throughout the software development lifecycle.

However, this new paradigm also provides unprecedented opportunities for the future of software development. For example, LLMs are set to revolutionize IDEs, enabling a collaborative ecosystem where humans and augmented intelligences (AI+) tools [Ozkaya et al. 2023] work together as trusted partners [Carleton et al. 2022]. This collaboration not only promises to accelerate the development and maintenance of software-reliant systems but also lowers the barrier to entry for computational thinking, making it accessible to a broader range of individuals with varying levels of formal education in programming [Diamandis 2024].

Motivating the need for prompt engineering. Programming first-generation LLMs, including (but not limited to) ChatGPT [Bang et al. 2023], involves natural language prompts, such as asking an LLM to explain a software vulnerability or generate JavaScript for a web page. These simple examples of prompts, however, only hint at the

significantly more sophisticated computational abilities of LLMs. Harnessing the potential of LLMs in productive ways requires a systematic focus on *prompt engineering* [Chen et al. 2023]. This emerging discipline studies structured interactions with—and programming of—LLM computational systems to solve complex problems via natural language interfaces.

This paper is an extensive revision of an earlier paper [White et al. 2023a] that described how pattern-oriented prompt engineering techniques can enhance the application of conversational LLMs for tasks in the software development lifecycle. Example applications of LLMs in this domain include helping developers code effectively and efficiently with unfamiliar APIs, allowing students to acquire new coding skills and techniques, and enabling cybersecurity professionals to rapidly detect and thwarts cyber-attacks [Gennari et al. 2024].

To demonstrate the promise of prompt engineering in the context of the software development lifecycle, we provided the following prompt to ChatGPT:

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

Prompts can also be engineered to program an LLM to do more than simply dictate the output type or filter the information provided to the LLM. With the right prompt it is possible to create entirely new interaction paradigms. For example, an LLM can generate and give a quiz associated with a software engineering concept or tool. Likewise, it can simulate a Linux terminal on a computer that’s been compromised by a cyber-attack (see Section 6.2 for more discussion on this use case).

Prompts also have the potential for self-adaptation. For example, a prompt can suggest other prompts to gather additional information or generate related artifacts (see Section 9 for more discussion on these use cases). These advanced capabilities of prompts highlight the importance of engineering them systematically to provide value beyond simple generation of text, code, or unit tests.

Prompt patterns are an essential foundation to an effective discipline of prompt engineering. A key contribution of this paper is codifying successful approaches for systematically engineering different input, output, and interaction behaviors when working with conversational LLMs via *prompt patterns*. Prompt patterns are similar to software patterns [Gamma et al. 1995; Schmidt et al. 2013] since they both offer reusable solutions to problems arising within particular contexts. The contexts they focus on, however, relate to interactions with LLMs, such as (but again not limited to) ChatGPT.

By documenting and leveraging prompt patterns in the context of tasks performed within the software development lifecycle, individuals and teams can enforce constraints on input formats and generated output to ensure relevant information is included. Likewise, prompt patterns can modify and adapt interactions with LLMs to better solve software-related problems. Prompt patterns can be viewed as a corollary to the broad corpus of software patterns, but are adapted to specific contexts of LLM interactions.

This paper focuses on domain-independent prompt patterns and presents a catalog of such patterns that have been applied to simplify and/or automate tasks in the software development lifecycle. Examples of such tasks include generating visualizations, code artifacts, and test cases; automating output steps for code editing; and identifying discrepancies between software requirements specifications and regulatory documents. Just as catalogs of software patterns [Gamma et al. 1995; Schmidt et al. 2013] codify ways of enhancing common software development tasks, catalogs of prompt patterns codify ways of enhancing LLM inputs, outputs, and interactions.

Paper organization. The remainder of this paper is organized as follows: Section 2 introduces prompt patterns and compares these patterns to classic software patterns; Section 3 summarizes and categorizes thirteen prompt patterns we identified to solve common problems in the domain of conversational LLM interaction and output

generation for tasks in the software development lifecycle; Section 4 describes a prompt pattern that controls the contextual information on which LLMs operate; Section 5 describes a prompt pattern that dictates how LLMs understand user input and how they translate this input into formats they can use to generate output; Section 6 describes prompt patterns that constrain or tailor the types, formats, structures, and/or other properties of the output generated by LLMs; Section 7 describes a prompt pattern that identifies and resolves errors in the output generated by LLMs; Section 8 describes prompt patterns that improve the quality of LLM input and output; Section 9 describes prompt patterns that manage the interaction between users and LLMs; Section 10 discusses related work; Section 11 presents concluding remarks and lessons learned; and Appendix A explains our approach for defining a prompt pattern’s structure and key ideas.

2. COMPARING SOFTWARE PATTERNS WITH PROMPT PATTERNS

The quality of the output(s) generated by the first-generation of LLMs is highly related to the quality of the prompts provided by users. As discussed in Section 1, the prompts given to a conversational LLM can program interactions between users and the LLM to solve a diverse set of problems effectively and efficiently. This section presents a framework for documenting patterns that structure prompts to solve a range of software-related tasks that can be adapted to different domains.

This framework is useful since it codifies patterns that help users interact more effectively with conversational LLMs in a variety of contexts, rather than simply showing selected examples or domain-specific prompts. Codifying this knowledge in pattern form enhances reuse and transferability to other contexts and domains where users face similar—though not necessarily identical—problems. This type of knowledge transfer has been studied in the software patterns literature at multiple levels, *e.g.*, design [Gamma et al. 1995], architecture [Schmidt et al. 2013], and analysis [Fowler 1996]. This paper applies a variant of a classic pattern form as the basis of our prompt engineering approach. Since prompts are a form of programming (albeit programming LLMs via natural language), it is natural to document them in pattern form.

2.1 Overview of Software Patterns

A software pattern provides a reusable solution to a recurring problem within a particular context [Gamma et al. 1995]. 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 [Buschmann et al. 2007]. Readers familiar with classic software patterns can skip ahead to Section 2.2.

A pattern form also includes guidance on how to implement a pattern, as well as information on the trade-offs and considerations to take into account when applying a pattern. Moreover, example implementations of a pattern are often provided to further showcase its applicability in practice. To facilitate their understanding and use, software patterns are often documented in a stylized form, such as the “Gang of Four” [Gamma et al. 1995] pattern form described below:

- A name and classification.** Each pattern has a name that identifies the pattern and should be used consistently. Software patterns can be classified in various ways, including purpose (*.e.*, creational, structural, or behavioral patterns), granularity (*e.g.*, design, architectural, or enterprise patterns), etc.
- The intent** concisely conveys the purpose the pattern is intended to achieve.
- The motivation** documents the underlying problem and “forces” the pattern is meant to address and underscores the importance of the problem.
- The structure and participants.** The structure describes key pattern participants (such as subsystems, classes, objects, and/or functions) and depicts how they collaborate to form a generalized solution.
- Example code** concretely maps the pattern to some concrete programming language(s) to help developers gain insight on how to apply the pattern effectively.

—**Consequences** summarize considerations users should take into account when deciding whether or how to apply this pattern.

2.2 Overview of Prompt Patterns

Prompt patterns serve a similar purpose and follow a similar format to software patterns outlined above, 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. Prompt patterns can be classified in various ways, such as the categories of pattern types summarized in the table in Section 3.
- The context and intent** concisely describe the purpose the prompt pattern is intended to achieve in a particular context. The context should ideally be domain independent, though domain-specific patterns can be documented with an appropriate discussion of the context where the pattern applies.
- The motivation** provides the rationale for the underlying problem and explains why solving it is important. The motivation is explained in the context of users interacting with a conversational LLM and how the pattern can improve upon users informally prompting the LLM in one or more circumstances. Specific circumstances where these improvements are expected are documented.
- The structure and key ideas.** The structure describes the fundamental contextual information² (presented as one or more key ideas) that users applying the prompt pattern provide to an LLM. These key ideas are similar to “participants” in a software pattern.
- Example implementation** demonstrates how the prompt pattern can be worded in practice and outlines various types of output generated by an LLM.
- Consequences** summarize considerations users should take into account when deciding whether or how to apply this pattern, as well as provides guidance on how to adapt the prompt to different contexts.

3. SUMMARY OF OUR CATALOG OF PROMPT PATTERNS FOR CONVERSATIONAL LLMs

This section summarizes our catalog of thirteen prompt patterns shown in the adjacent table, which also indicates the page number where the prompt pattern description starts. These prompt patterns have been applied in the domain of conversational LLMs to solve common problems related to tasks performed within the software development lifecycle. Although these prompt patterns can be generalized to domains beyond software development, we focus largely on that domain in this paper to make the examples more concrete and relevant for our intended audience.

Organizing our catalog of prompt patterns into the smaller number of categories shown in this table helps users apply these patterns more effectively. The following are the six categories of prompt patterns in our classification framework:

Pattern Category	Prompt Pattern	Page
Context Control	<i>Context Conveyor</i>	6
Input Semantics	<i>Meta Language Creation</i>	8
Output Customization	<i>Output Automater</i>	9
	<i>Persona</i>	11
	<i>Visualization Generator</i>	13
	<i>Recipe</i>	14
	<i>Output Template</i>	16
Error Identification	<i>Fact Check List</i>	17
Prompt Improvement	<i>Alternative Approaches</i>	19
	<i>Question Decomposition</i>	20
	<i>Refusal Breaker</i>	22
Interaction	<i>Flipped Interaction</i>	24
	<i>Infinite Generation</i>	25

¹The most direct translation of software pattern structure to prompt patterns is the naming, intent, motivation, and sample code. The structure and classification require more adaptation, however, although we named them similarly to abide by the “principle of least surprise.”

²Appendix A describes how and way we use fundamental contextual statements to convey key ideas contained in a prompt to communicate with an LLM.

- The **Context Control** category focuses on controlling the contextual information that LLMs operate upon, which is critical to ensure coherent, relevant, and accurate responses from LLMs.
- The **Input Semantics** category focuses on how LLMs understand user input and how they translate this input into formats they can use to generate output.
- The **Output Customization** category focuses on constraining or tailoring the types, formats, structures, and/or other properties of the output generated by LLMs.
- The **Error Identification** category focuses on identifying and resolving errors in the output generated by LLMs.
- The **Prompt Improvement** category focuses on improving the quality of LLM input and output.
- The **Interaction category** focuses on the dynamics between users and LLMs.

We identified these six categories and thirteen prompt patterns during our initial work with ChatGPT. Other categories and prompt patterns we identified to improve code quality, refactoring, requirements elicitation, and software design appear in [White et al. 2023b].

The remainder of this paper presents each of the thirteen prompt pattern using the pattern form described in Section 2.2. Each prompt pattern is accompanied by concrete implementation examples. These examples were obtained through a combination of exploring the corpus of community-posted prompts on the Internet and independent prompt creation through our application of ChatGPT to help automate tasks throughout the software development lifecycle.

All prompts and examples in this paper were tested with ChatGPT [OpenAI 2023a] due to its widespread availability, popularity, and capabilities. Although the output from ChatGPT has been limited or omitted for brevity in most cases, we encourage readers to use ChatGPT to test the prompt patterns documented below. These patterns are all readily testable using any version of ChatGPT, though we recommend using ChatGPT-4 since it is more powerful and reliable than ChatGPT-3.5.

4. PROMPT PATTERNS FOR CONTEXT CONTROL

The **Context Control** category focuses on managing the contextual information on which LLMs operate. This section describes the *Context Conveyor* pattern, which allows users to specify the context for an LLM's output. Providing sufficient context is critical to ensure that an LLM retains and utilizes relevant information throughout the course of a user's interaction, thereby enabling more coherent, relevant, and accurate responses [Zhu et al. 2024].

4.1 The *Context Conveyor* Pattern

Context and Intent. Users may need greater control over the context an LLM considers or ignores when generating output. The intent of the *Context Conveyor*³ pattern is to enable users to specify or remove context for a conversation with an LLM to help focus the conversation on specific topics and/or exclude unrelated topics from consideration.

Motivation. Context is crucial for LLMs because it enables them to generate more accurate, relevant, and coherent responses by understanding the broader scope and nuances of a conversation. LLMs often struggle, however, to interpret the intended context of the current question or they can generate irrelevant responses based on prior inputs or irrelevant attention on the wrong statements. For example, users may introduce unrelated topics or reference information from earlier in a dialogue, which may can disrupt the flow of the conversation with an LLM.

The *Context Conveyor* pattern either emphasizes or removes specific aspects of the context to maintain relevance and coherence in user conversations with LLMs. By focusing on explicit contextual statements—or removing irrelevant statements—this pattern enables users to help an LLM better understand their questions so it generates more accurate responses.

³The *Context Conveyor* pattern was called the *Context Manager* pattern in [White et al. 2023a]. We changed the name of this pattern in this paper to better reflect its intent.

Structure and Key Ideas. The fundamental contextual statements associated with the *Context Conveyor* pattern are shown in the table below:

Contextual Statements
1. Within scope X
2. Please consider Y
3. Please ignore Z
4. (Optional) start over

Statement #1 helps to scope the context, such as “when performing a code review” or “when generating a deployment script,” to establish the boundaries within which certain information should be considered. The more explicit these statements are about the context, the more likely the LLM will take appropriate action. For example, focusing on modularity may be important for a code review but not for a deployment script.

Statements #2 and #3 describe information to incorporate into the output. Statements about what to consider or ignore should list key concepts, facts, instructions, etc. that should be included or removed from the context. For example, if users ask an LLM to ignore subjects related to a topic—yet some of 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 LLM’s inclusion/exclusion behavior will be.

Finally, statement #4 optionally instructs an LLM to explicitly “start over.” This statement can be added at the end of a prompt if the goal is to wipe the slate clean. A better way accomplish this task, however, may be to simply start a new interaction session with the LLM.

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 LLMs so they can better understand the intended scope of the conversations and generate more relevant responses. In many situations, users may want to completely start over and can employ this prompt to reset LLM context:

“Ignore everything that we have discussed and start over.”

The “start over” statement instructs the LLM to produce a complete reset of the context.

Consequences. The following are a summary of considerations users should take into account when deciding whether or how to apply the *Context Conveyor* pattern.

Unintentional reset risks. An organization may transparently inject a series of policy filters or helpful prompts into the start of each LLM conversation to restrict and/or enhance conversations. However, users may not be aware of these filters/prompts. Applying this pattern may therefore inadvertently reset or eliminate previously applied filters or prompts within a conversation, potentially diminishing the LLM’s guardrails or functionality without users’ knowledge.

Solution strategies for context preservation. Users should be made aware of existing contexts and the potential consequences of resets. It may therefore be useful to incorporate strategies to inform users about what might be lost before any context alteration occurs, such as integrating prompt explanations or confirmations that summarize the impacts of context changes. These strategies provide a safeguard against unintentional functionality loss and help ensure users make informed decisions about altering the conversation’s context.

5. PROMPT PATTERNS FOR INPUT SEMANTICS

The **Input Semantics** category focuses on how LLMs understand user input and how they translate this input into formats they can use to generate output. This section describes the *Meta Language Creation* pattern, which

creates a custom language for an LLM to understand. Users should consider applying this pattern when the default input language and/or format is ill-suited for expressing ideas they want to convey to the LLM.

5.1 The *Meta Language Creation* Pattern

Context and Intent. During conversations with an LLM, users may want to write prompts using an alternate notation or language, such as a textual short-hand notation for graphs, a description of states and state transitions for a state machine, or a set of commands for prompt automation. The intent of the *Meta Language Creation* pattern is to explain the syntax and semantics of an alternative language to an LLM so users can write subsequent prompts in this language and the LLM will understand what the prompts mean.

Motivation. Many problems, structures, or other ideas communicated in a prompt may be more concisely, unambiguously, or clearly expressed in a notation or language other than English (or whatever conventional human language is used to interact with an LLM). For example, a user might want to use the notation “ $a \rightarrow b$ ” to express an edge connecting two nodes in a directed graph. To produce output based on an alternative language, an LLM must understand the language’s syntax and semantics in terms of a language that it was trained on.

Structure and Key Ideas. A fundamental contextual statement associated with the *Meta Language Creation* pattern is shown in the table below:

Contextual Statements
1. When I say X, I mean Y
2. When I say X, I want you to do Y

The structure and key ideas of this pattern involves explaining the meaning of one or more symbols, words, or instruction to an LLM so it uses the given syntax and semantics to guide the ensuing conversation. Statement #1 above shows that this explanation can take the form of a simple translation, such as “When I say X, I mean Y.”

Statement #2 shows a more complex form that defines a series of commands and their semantics, such as “when I say X, I want you to do Y.” In this case, the LLM should henceforth bind “X” to the semantics of “take action Y.” For example, “whenever I type the word ‘outline’ print an outline of all the topics we’ve discussed thus far.”

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

“From now on, whenever I type two identifiers separated by a “ \rightarrow ,” I am describing a graph. For example, “ $a \rightarrow b$ ” is describing a graph with nodes “a” and “b” and an edge between them. If I separate identifiers by “ $-[w:2, z:3]\rightarrow$,” 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 the convention for representing nodes and edges. Whenever a user subsequently types two identifiers separated by a “ \rightarrow ” symbol, it indicates a graph is being described. For example, if a user types “ $a \rightarrow b$ ”, 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 and functionality of a graph to an LLM in textual 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 “ $-[w:2, z:3]\rightarrow$.” This notation allows the specification of additional properties beyond the basic structure of the graph. These specified properties are associated with the edge between the two nodes and can provide important context for the interpretation of the graph. This standardized notation for describing graphs makes it easier to communicate graph structures and properties, which may be complicated or overly verbose to describe as a series of natural language sentences.

Consequences. The following are a summary of considerations users should take into account when deciding whether or how to apply the *Meta Language Creation* pattern.

Ambiguity risks. Although this pattern can customize user interactions with an LLM, it may create the potential for confusion within the LLM itself. For example, the prompt “whenever I separate two things by commas, it means that the first thing precedes the second thing” will likely create significant potential for ambiguity and create the potential for unexpected semantics because commas are used as a standard form of punctuation that lack the semantic “a precedes b.” While it is important to clearly define the language’s syntax and semantics, it is also essential to ensure the language itself introduces no ambiguities that degrade an LLM’s performance or accuracy.

To showcase the potential for ambiguity and confusion, ChatGPT warns users and potentially may not 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 by ChatGPT’s output 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.”

Training data compatibility and contextual scoping. LLMs generally perform better on data they were trained on. Therefore, when 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 accuracy and utility of the output. For example, the symbol “→” conveys slightly different information in graphs and propositional logic. The statement “ $p \rightarrow q, q \rightarrow p$ ” may therefore make sense, but only if the LLM is given the appropriate scoping via the *Meta Language Creation* prompt pattern. It may also be helpful to combine this pattern with the *Context Conveyor* pattern (page 6) to set the context explicitly.

Best practices for implementation. Although the *Meta Language Creation* pattern is powerful it must be used carefully, particularly when describing concepts that may otherwise be hard to describe precisely or concisely. These types of prompts are thus best used in completely new conversation sessions with an LLM. Using a single meta-language-per-conversation session is also a best practice to avoid the potential for an LLM to apply conflicting or unexpected semantics to a conversation over time.

6. PROMPT PATTERNS FOR OUTPUT CUSTOMIZATION

The **Output Customization** category focuses on constraining or tailoring the types, formats, structures, or other properties of the output generated by LLMs. Prompt patterns presented in this section include *Output Automater*, *Persona*, *Visualization Generator*, *Recipe*, and *Output Template* patterns.

The *Output Automater* pattern creates scripts that automate tasks the output of an LLM suggests users should perform. The *Persona* pattern gives an LLM a role to play when generating output. The *Visualization Generator* pattern allows users to generate visualizations by producing textual outputs that can be fed to other tools, such as AI-based image generators, like DALL-E [OpenAI 2023b]. The *Recipe* pattern allows users to obtain a sequence of steps or actions to realize a stated outcome, possibly with partially-known information or constraints. The *Output Template* pattern allows users to specify a template for the output, which an LLM then fills in with content.

6.1 The *Output Automater* Pattern

Context and Intent. Users may want to reduce the manual effort needed to implement any output recommendations provided by an LLM. The intent of the *Output Automater* pattern is to have an LLM generate automation artifact (such as a Python script) that can automatically perform any steps it recommends as part of its output. This pattern also allows an LLM to execute actions in other systems, such as updating databases, sending emails, or managing tasks in project management tools.

Motivation. The output of an LLM is often a sequence of steps for a user to follow. For example, when asking an LLM to generate a Python configuration script it may suggest a number of files to modify and changes to apply to each file. Another output from an LLM may be a sequence of configuration actions in a cloud computing platform, such as the Amazon Web Services console.

It can be tedious and error-prone, however, for users to repeatedly perform the manual steps dictated by LLM output. The *Output Automater* pattern enables seamless interaction between an LLM and a range of external applications and services. This pattern extends an LLM's utility beyond mere information generation to directly facilitate actionable outcomes.

Structure and Key Ideas. The fundamental contextual statements associated with the *Output Automater* pattern are shown in the table below:

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

Statement #1.A identifies the situations under which automation should be generated. A simple approach is to state the output includes at least 'X' steps to perform and that the LLM should produce an automation artifact. This “scope limiting” helps prevent an LLM from producing output automation scripts in cases where running such scripts requires more user effort than simply performing the original steps produced in the output. In particular, the scope can be limited to outputs requiring more than a given number of steps.

Statement #1.B refines statement #1.A by providing concrete guidance on the type of output an LLM should generate to perform the automation. For example, “produce a Python script” gives the LLM a concrete understanding to translate the general steps into equivalent steps in Python. The automation artifact should be concrete and something the LLM associates with the action of “automating a sequence of steps.”

Example Implementation. A sample of the *Output Automater* pattern applied to code snippets generated by ChatGPT 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 sample prompt automates the common software development task of taking the LLM output and editing one or more files. The scope is “whenever you generate code that spans more than one file,” which occurs when developers would have multiple—potentially error-prone steps—to make the necessary code edits. The “generate a Python script that can be run to automatically” portion of this prompt instructs the LLM to automate file editing instead of having developers do this manually. The specification of the goal is important to prevent the LLM from automating unrelated tasks since the output may have other steps to complete that are unrelated to coding.

Consequences. The following are a summary of considerations users should take into account when deciding whether or how to apply the *Output Automater* pattern.

Enhancing automation in computing systems. This pattern is a powerful complement for any computer-controlled system, such as

- Generating development operations (DevOps) artifacts for build, configuration and deployment,
 - Code editing tasks, including the creation and editing of multiple files, and
 - Environment setup tasks, ranging from creation of environment variables to running necessary shell commands.
- An LLM can provide a set of steps to perform on the computer-controlled system and the output can then be translated into a script that instructs the computer controlling the system to perform these steps automatically. This pattern enables LLMs to integrate quality into—and exert control over—new computing systems that have a well-defined scripting interface.

Scope of applicability. The *Output Automater* pattern is particularly effective in software development tasks where programmers use LLMs to generate output that they then copy/paste into multiple files. Some AI-assisted tools, such as Microsoft Copilot, insert limited snippets directly into the section of code on which a programmer is working. In contrast, conversational LLMs, such as ChatGPT, do not provide such facilities. This prompt pattern is also effective at creating scripts for running commands on a terminal, automating cloud operations, and/or reorganizing files and folders in a file system.

Definition clarity. 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, an LLM often states that it “can’t automate things” since that is beyond its capabilities. LLMs typically accept requests to produce code, however, so this prompt pattern instructs the LLM to generate text/code, which can be executed to automate something. This subtle distinction in meaning is important to help an LLM disambiguate the meaning of a prompt based on the *Output Automater* pattern.

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

Handling omissions. An LLM may produce a long output with multiple steps and not include an automation artifact. This omission can 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 it with the context that the automation artifact was omitted and should be generated.

User responsibility. 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 offloads users from performing certain manual steps, it does not alleviate their responsibility for understanding the actions they undertake using the LLM’s output. When users execute automation scripts they therefore assume responsibility for the outcomes.

6.2 The *Persona* Pattern

Context and Intent. Users may want output from an LLM to always take a certain point of view or perspective. The intent of the *Persona* pattern is to instruct the LLM the play a role that helps it select what details to focus on and what types of output to generate.

Motivation. Users may not know precisely what types of outputs or details are important for an LLM to focus on to achieve a given task. They may know, however, the role or type of person they would normally ask for help performing these tasks. For example, it may be useful to conduct a code review as if the LLM was a security expert, even if users themselves are not security experts. The *Persona* pattern enables users to express what they need help with, without knowing the exact details of the outputs they need.

Structure and Key Ideas. The fundamental contextual statements associated with the *Persona* pattern are shown in the table below:

Contextual Statements
1. Act as persona X
2. Provide outputs that persona X would create

Statement #1 instructs an LLM to act as a specific persona and provide outputs like such a persona would. This persona can be expressed in a number of ways, including a job description, title, fictional character, or historical figure. The persona should elicit a set of attributes associated with a well-known job title, type of person, etc.

Statement #2 offers opportunities for customization by instructing the LLM to provide outputs that someone with persona X would create. For example, a computer science teacher might provide a large variety of different output types, ranging from programming assignments to reading lists to lectures. If more specific scope(s) to the type(s) of output(s) are known, users can provide them as part of the contextual statement.

Example Implementation. A sample implementation for applying the *Persona* pattern in a 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 regard regarding the code.”

In this example, the LLM is instructed to provide outputs that a “security reviewer” would. The prompt further sets the context within which the code will be evaluated. Finally, the user refines the prompt by scoping the persona further to outputs regarding the code’s security posture.

Personas can also represent inanimate or non-human entities, such as a Linux terminal, a database, or even an animal’s perspective. When using this pattern to represent these types of entities, it can be useful to specify how the inputs will be 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 will pretend to be a Linux terminal for a computer that has been compromised by an attacker. When I type in a command, you will 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 first input commands into the terminal. In response, the simulated terminal will output the corresponding text that would be produced by a real Linux terminal. This prompt is more prescriptive in the persona and asks the LLM to not only be a Linux terminal, but to act as a computer compromised by an attacker.

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

Consequences. The following are a summary of considerations users should take into account when deciding whether or how to apply the *Persona* pattern.

Assumptions and hallucinations. An interesting aspect of instructing an LLM to taking on non-human personas is that it may make interesting assumptions or “hallucinations” regarding the context. For example, ChatGPT can be instructed to act as a Linux terminal and generate the output expected if a user typed the same text into a terminal. Henceforth, commands like `ls -l` will generate a file listing for an imaginary UNIX file system, complete with files that can have other Linux commands like `cat file1.txt` run on them.

Contextual prompts for realism. An LLM may prompt the user for more context, such as when ChatGPT is asked to act as a MySQL database and thus prompts the user for the structure of a table the user is pretending to query. ChatGPT can then generate synthetic rows, such as generating imaginary rows for a “people” table with columns like “name” and “job.”

Persona limitations and policy filters. When personas are based on real individuals, privacy issues become paramount. LLMs must navigate the fine line between providing engaging, persona-driven content and respecting

the privacy and consent of the individuals being represented. An LLM may therefore disregard requests for personas relating to living people or people considered harmful due to underlying policy filters, such as privacy and security rules.

6.3 The *Visualization Generator* Pattern

Context and Intent. Many concepts are easier to grasp in diagram or image format. The intent of the *Visualization Generator* pattern is to use text generation to create visualizations.

Motivation. The motivation for this pattern is to enhance the output of an LLM so it can leverage external visualization generators for images that the model may not be able to accurately generate directly. LLMs generally produce text or images using image generation models that may not be suited for all tasks, such as drawing scientific diagrams. For example, an LLM cannot often draw a diagram to describe a graph of a complex concept.

The *Visualization Generator* pattern overcomes limitations with LLMs by generating textual inputs in the correct format to plug into another tool that generates the correct diagram. For example, users can apply this pattern to create visualizations by creating inputs for other well-known visualization tools that use text as their input, such as Graphviz Dot [Ellson et al. 2004] or DALL-E [OpenAI 2023b]. By using text inputs to generate visualizations, users can quickly understand complex concepts and relationships that may be hard to grasp through text alone.

Structure and Key Ideas. The fundamental contextual statements associated with the *Visualization Generator* pattern are shown in the table below:

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

Statement #1 indicates to an LLM that the output it produces (*i.e.*, “X”) will be some type of image or visualization. When an LLM can’t generate a particular type of image or visualization natively, the portion of statement #1 that says “that I can provide to tool Y to visualize it” clarifies that the LLM is not expected to generate an visualization *per se*, but instead is expected to produce a description of a visualization consumable by tool Y for production of the visualization.

Many tools may support multiple types of visualizations or formats, and thus naming the target tool itself may provide insufficient information to accurately produce what the user wants. Users may therefore 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. As will be discussed in example implementation below, it can be advantageous to specify a list of possible tools and formats and let the LLM select the appropriate target for visualization.

Example Implementation. A sample implementation for applying the *Visualization Generator* pattern in a code review is shown below:

“Whenever I ask you to visualize something, please create either a Graphviz Dot file or Vega-lite specification that I can use to create the visualization. Choose the appropriate tools based on what needs to be visualized.”

This example adds a qualification that the output type for the visualization can be either for Graphviz or Vega-lite. This approach allows an LLM to use its semantic understanding of the output format to automatically select the target tooling based on what will be displayed.

In this example, Graphviz would be applied to visualize graphs with a need for an exactly defined structure. Vega-lite would be effective at visualizing graphs, charts, and more data-driven visualizations. An LLM can select the tool based on the needs of the visualization and capabilities of each tool.

Consequences. The following are a summary of considerations users should take into account when deciding whether or how to apply the *Visualization Generator* pattern.

Enhanced communication through hybrid text/visualization outputs. This pattern creates a pathway for the tools to produce visualizations that are associated with other outputs. This pattern can provide a more comprehensive and effective way of communicating information by combining the strengths of both the text generation and external visualization tools.

Expanded expressive capabilities. The pattern creates a target pipeline for the output to render a visualization. The pipeline may include AI generators, such as Midjourney, that can produce rich visualizations but are provided by other vendors. The pattern allows the user to expand the expressive capabilities of the output into the visual domain.

Dependency on third-party visualization tools. One drawback of applying the *Visualization Generator* pattern is its reliance on external visualization tools and AI generators like Midjourney, which introduces dependencies on third-party tools that can lead to issues of consistency, reliability, and control over the visualizations.

Consistency and quality control challenges. While this pattern enriches the expressive capabilities of the output by extending it into the visual domain, it also complicates the process by incorporating tools that may have varying standards of quality, different operational limitations, and distinct licensing requirements. These factors can impact the seamless integration of textual and visual information, potentially hindering the overall effectiveness and accessibility of the generated content.

6.4 The *Recipe* Pattern

Context and Intent. Users often want an LLM to analyze a concrete sequence of steps or procedures to achieve a stated outcome. The intent of the *Recipe* pattern is to provide constraints intended to ultimately output a sequence of steps given some partially provided “ingredients” that must be configured in a sequence of steps to achieve a stated goal.

Motivation. Users often generally know—or have an idea of—what the end goal of a prompt 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.”

The *Recipe* pattern generalizes 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. For example, “provide three different ways of deploying a web application to AWS using Docker containers and Ansible using step-by-step instructions.”

Structure and Key Ideas. The fundamental contextual statements associated with the *Recipe* pattern are shown in the table below:

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

Statement #1 focuses the LLM on the overall goal (“X”) the recipe must be built to achieve. The LLM will organize and complete these steps to achieve the specified goal sequentially.

Statement #2 provides the partial list of steps (“A, B, C”) the user would like the LLM to include in the overall recipe. These steps serve as intermediate waypoints for the path the LLM may take to generate or constraints on the structure of the recipe.

Statement #3 indicates to the LLM that the goal is to provide a complete sequential ordering of steps.

Statement #4 helps ensure the LLM will attempt to complete the recipe without further follow-up by making some choices on the user's behalf regarding missing steps, as opposed to just stating additional information that is needed.

Statement #5 helps flag inaccuracies in the user's original request so the final recipe is efficient.

This pattern combines elements of the *Output Template*, *Alternative Approaches*, and *Question Decomposition* patterns, as follows:

- The *Recipe* pattern borrows the structured output format from the *Output Template* pattern (page 16) by organizing the information provided by the user into a coherent sequence of steps. By applying the *Output Template* pattern, the *Recipe* pattern ensures that the LLM produces outputs in a predictable, structured format that aligns with the user's request for a "complete sequence of steps," thereby facilitating easier interpretation and implementation of the solution.
- From the *Alternative Approaches* pattern (page 19), the *Recipe* pattern adopts the flexibility of exploring different methods to achieve the stated goal ("X"), which is crucial when the initial steps provided ("A, B, C") may not be the only or the most efficient path to the desired outcome. The inclusion of this element allows an LLM to consider and possibly suggest alternative strategies or steps that might not have been initially apparent or provided by the user, enhancing the recipe's effectiveness and adaptability.
- The *Recipe* pattern utilizes the *Question Decomposition* pattern's (page 20) approach to break down the overall goal into smaller, manageable tasks. By identifying and filling in missing steps (Statement #4) and eliminating unnecessary ones (Statement #5), the LLM effectively decomposes the complex problem ("achieve X") into a series of simpler tasks. This decomposition aids in the logical sequencing of steps and ensures that the final recipe is both complete and optimized for efficiency.

By incorporating elements from these other three patterns, the *Recipe* pattern can deliver a comprehensive and efficient solution by structuring the output in a clear format, considering multiple pathways to the goal, and breaking down the task into simpler components for better clarity and execution.

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

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

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

Consequences. The following are a summary of considerations users should take into account when deciding whether or how to apply the *Recipe* pattern.

Specificity of user input. When users apply this pattern, the effectiveness of an LLM's output heavily depends on the clarity and detail of user initial descriptions. If these descriptions are vague or lack specificity, LLMs might generate solutions that are overly general, missing critical nuances of the given task. Although this output may technically address the request, it may not fully capture actual needs or intentions of users. In contrast, a well-specified input can guide an LLM towards generating more tailored and applicable solutions, underscoring the importance of precise and detailed user input when applying this pattern.

Introduction of bias. The initial steps or requirements outlined by users when applying the *Recipe* pattern can inadvertently introduce bias into an LLM's processing and output. This bias may manifest in the LLM prioritizing

solutions that align with the specifics of the user-provided steps, even when those steps are not necessary or optimal for the task.

For instance, if users specify certain dependencies or tools in their requests, an LLM might focus on solutions that incorporate these elements, potentially overlooking simpler or more efficient alternatives. This consequence highlights the need for careful consideration of the initial input provided to an LLM since it can significantly influence the direction and nature of the solution generated by the LLM.

6.5 The *Output Template* Pattern

Context and Intent. User may want to instruct an LLM to produce output in a format it would not ordinarily use for the specified type of content being generated. The intent of the *Output Template*⁴ pattern is to ensure the structure of an LLM's output follows a precise format.

Motivation. In some cases, LLM output must be produced in a precise format that is specific to a particular application or use case, but which is unknown to the LLM. For example, users might need to generate URLs that insert generated information into specific positions within URL paths. If an LLM is unaware of this template structure, it must be instructed what the format is and where the different parts of its output should go. These instructions could take the form of a sample data structure to generate, a series of form letters being filled in, etc.

Structure and Key Ideas. The fundamental contextual statements associated with the *Output Template* pattern are shown in the table below:

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

Statement #1.A directs the LLM to follow a specific template for its output. This template will be used to coerce the LLMs responses into a structure consistent with user formatting needs, which is useful when the LLM does not know the target format. If the LLM already has knowledge of the format (such as a specific file type), the *Output Template* pattern can be skipped and the user can simply specify the known format. However, there may be cases, such as generating Javascript Object Notation (JSON), where large variation exists in how data could be represented within a format. In such cases the *Output Template* pattern helps ensure the LLM's output meets additional user constraints specified by the target format.

Statement #1.B makes the LLM aware that the template contains a set of placeholders that enable users to explain how the output should be inserted into the template. They also allow users to target where information should be inserted semantically. Placeholders can use formats (*e.g.*, NAME) that allow an LLM to infer semantic meaning and determine where output should be inserted (*e.g.*, insert the person's name in the NAME placeholder). Placeholders also enable users to indicate what is *not* needed in the output, *e.g.*, if a placeholder does not exist for a component of the generated output that component can be omitted. Ideally, placeholders should use a format (*e.g.*, all caps or enclosure in brackets) commonly employed in text the LLM was trained on.

Statements #1.C and #1.D constrain the LLM so it does not arbitrarily rewrite the template or attempt to modify it so all the output components can be inserted. These statements, however, may not preclude an LLM from generating additional text generated before or after. In practice, LLMs typically follow the template, but it may be

⁴The *Output Template* pattern was called the *Template* pattern in [White et al. 2023a]. We changed the name of this pattern in this paper to better reflect its intent.

hard to eliminate additional text being generated beyond the template without further experimentation with prompt wording.

Finally, statement #1.E provides the actual template the LLM should output its response in, such as

Email Reply Template

Subject: RE: [Subject Placeholder]

Dear [Recipient Name Placeholder],

Thank you for your [Reason for Email Placeholder].

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

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

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

User: “Generate a name and job title for a person”

ChatGPT: “https://myapi.com/Emily_Parker/profile/ Software_Engineer”

Consequences. The following are a summary of considerations users should take into account when deciding whether or how to apply the *Output Template* pattern.

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

Compatibility with other patterns. Filtering may also make it hard to combine this pattern with other patterns from the **Output Customization** category. The *Output Template* pattern effectively constrains the output format, so it may not be compatible with generation of certain other types of output. For example, in the template provided above for a URL, it would not be easy (or even possible) to combine the *Output Template* pattern with the *Recipe* pattern (page 14), which needs to output a list of steps.

7. PROMPT PATTERNS FOR ERROR IDENTIFICATION

The **Error Identification** category focuses on identifying and resolving errors in the output generated by LLMs. This section describes the *Fact Check List* pattern, which initially requires an LLM to generate a list of facts the output depends on that should be fact-checked. It then instructs the LLM to introspect on its output to identify errors it can correct.

7.1 The *Fact Check List* Pattern

Context and Intent. Users of LLMs are often (justifiably) wary of the output they generate since it is often not clear what facts (or assumptions) the output is based on. The intent of the *Fact Check List* pattern is to ensure the LLM outputs a list of facts that are present in the output and form an important part of the statements in the output. Users can then perform appropriate due diligence on these facts/assumptions to validate the veracity of the LLM's output.

Motivation. A current weakness of LLMs (including ChatGPT) is they often rapidly (and enthusiastically) generate text that sounds authoritative, but is actually factually incorrect. These errors can take a wide range of forms, ranging from fake statistics to invalid version numbers for software library dependencies. Due to the convincing

nature of the outputs generated by LLMs, however, users may not perform appropriate due diligence to determine output accuracy. The *Fact Check List* pattern helps to mitigate this problem by providing a structured method for users to systematically verify the accuracy of information generated by LLMs, encouraging critical evaluation and reducing the risk of accepting erroneous data as fact.

Structure and Key Ideas. The fundamental contextual statements associated with the *Fact Check List* pattern are shown in the table below:

Contextual Statements
1.A Generate a set of facts that are contained in the output
1.B The set of facts should be inserted in the output at [a specific point]
1.C The set of facts should be the fundamental facts that could undermine the veracity of the output if any of them are incorrect

Statement #1.A instructs an LLM to identify the facts that are contained within its output. The LLM should be able to identify facts effectively since they are a well-understood concept and not impacted by the actual content, *i.e.*, the concept of a “fact” is domain-independent.

Statement #1.B tells the LLM where the facts should be included in the output at a specific point. For example, the facts could be included at the end or beginning of the output. Of course, other arrangements could be employed.

Statement #1.C expresses the idea that facts should be the ones most important to the overall veracity of the statements, *i.e.*, choose facts fundamental to the argument and not derived facts flowing from those facts. This statement is crucial since it helps to scope the output to those facts most important to the veracity and not derived statements that may be less important. Of course, this constraint could also be relaxed.

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

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

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

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

Consequences. The following are a summary of considerations users should take into account when deciding whether or how to apply the *Fact Check List* pattern.

Domain-specific utility. This pattern should be employed whenever users are not experts in the domain for which they are generating output, which may occur when combining this prompt pattern with the *Persona* pattern (page 11). For example, software developers reviewing code may benefit from security consideration suggestions. In contrast, an expert on software architecture may identify errors in statements about the software structure and may not need a fact check list for these outputs.

Integration with other patterns. Errors are potential in all LLM outputs, so *Fact Check List* is an effective pattern to combine with other patterns, such as the *Question Decomposition* pattern (page 20). This combination

of prompt patterns enables a more comprehensive approach to verifying LLM outputs and enhancing the reliability of generated content.

Fact verification process. A key aspect of the *Fact Check List* pattern is that users can inherently check the list of facts against the output. In particular, users can directly compare a fact check list to the output to verify the facts in the fact check list actually appear in the output. Users can also identify any omissions from the list. Although a fact check list may also have errors, users often have sufficient knowledge and context to determine its completeness and accuracy relative to the LLM's output.

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

8. PROMPT PATTERNS PROMPT IMPROVEMENT

The **Prompt Improvement** category focuses on improving the quality of LLM input and output. Prompt patterns presented in this section include *Alternative Approaches*, *Question Decomposition*, and *Refusal Breaker* patterns.

The *Alternative Approaches* pattern requires an LLM to suggest alternative ways of accomplishing a user-specified task. The *Question Decomposition* pattern instructs an LLM to automatically suggest a series of subquestions for users to answer before combining the answers to the subquestions and producing an consolidated answer to the overall question. The *Refusal Breaker* pattern requires an LLM to automatically reword user questions when it refuses to produce an answer.

8.1 The *Alternative Approaches* Pattern

Context and Intent. Users may not be familiar with alternative approaches that require them to think carefully about what they are doing and may need help to determine if they have identified the best approach(es) to meet reach their goal(s). The intent of the *Alternative Approaches* pattern is to ensure an LLM offers alternative ways of accomplishing a task so users do not only pursue familiar approaches. This pattern may also inform users about alternative concepts for subsequent follow-up prompts.

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

Structure and Key Ideas. The fundamental contextual statements associated with the *Alternative Approaches* pattern are shown in the table below:

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

The initial portion of statement #1 ("within scope X") scopes the interaction to a particular goal, topic, or bounds on the questioning. The scope is the constraint(s) users place on alternative approaches. For example, the scope could be "for implementation decisions" or "for the deployment of the application." This scope ensures that any alternatives fit within the boundaries or constraints to which users must adhere.

The next portion of statement #1 (“if there are alternative ways to accomplish the same thing, list the best alternate approaches”) instructs an LLM to suggest alternatives. As with other prompt patterns, instruction specificity can be increased or can include domain-specific contextual information. For example, this 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 an LLM from suggesting alternatives that are inherently non-viable because they require too many changes to other parts of an application.

Statement #2 optionally adds decision making criteria to the analysis, which is needed when users are not aware of alternative approaches and may thus not be aware of why and when to choose an alternative. This statement ensures the LLM provides users with the necessary rationale for alternative approaches.

Statement #3 optionally instructs an LLM to include the original approach given by the user for completeness when evaluation the LLM’s output.

Statement #4 optionally helps eliminate users needing to copy/paste or enter in an alternative approach manually if one is selected.

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

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

This implementation of the *Alternative Approaches* pattern is specifically tailored for the context of software engineering and focuses on deploying applications to cloud services. The prompt is intended to intercept places where developers have made a cloud service selection without full awareness of alternative services that may be priced more competitively or are easier to maintain. This 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), as well as to compare and contrast the pros and cons of each approach.

Consequences. The following are a summary of considerations users should take into account when deciding whether or how to apply the *Alternative Approaches* pattern.

Versatility and application. This pattern is broadly applicable, being effective in its generic form across diverse tasks and challenges. Its adaptability allows users to explore multiple solutions to a problem, enhancing creativity and problem-solving skills by encouraging users to consider a range of strategies they may not have been aware of previously.

Domain-specific catalogs. Refinements of this pattern could be used to implement a standardized catalog of acceptable alternatives within a specific domain, thereby guiding users towards the most relevant and effective solutions. These catalogs could make it easier for users to navigate their options within a structured framework, minimizing an otherwise overwhelming number of choices and ensuring that the alternatives presented are vetted and viable.

Informed decision-making. By presenting users with an approved set of approaches—complete with insights into their respective advantages and disadvantages—the *Alternative Approaches* pattern plays a crucial role in facilitating informed decision-making. It educates users about the potential outcomes of their choices. It also empowers users to make decisions that are best aligned with their goals and to specific nuances of their situation, ultimately yielding more satisfactory outcomes.

8.2 The Question Decomposition Pattern

Context and Intent. LLMs may produce better output when user inquiries are refined by decomposing them into smaller, manageable questions, allowing for more focused responses that the LLM then integrates into a

comprehensive answer. The intent of the *Question Decomposition*⁵ pattern is to enable this approach, prompting the user and/or the LLM to dissect the main question into sub-questions, whose answers are then combined to improve the response to the initial inquiry.

Motivation. The motivation for the *Question Decomposition* pattern emerges from the following two considerations aimed at enhancing user interactions with LLMs:

- Individuals frequently pose questions that are too abstract or high-level to elicit a specific answer directly. This issue may arise from various factors, such as a lack of familiarity with the subject matter, a tendency to minimize effort in formulating the question, or uncertainty regarding how to accurately phrase their inquiry. These initial, broadly framed questions often necessitate multiple iterations, where users provide additional clarifications or follow-up questions to LLMs, thereby narrowing down the scope and focusing on the underlying information needed by an LLM.
- Research [Zhou et al. 2022b] has shown that LLMs tend to produce more accurate and relevant responses when they refine a question by decomposing it into several more targeted questions. This refinement process allows an LLM to tackle each aspect of the main question in a more focused manner, leading to a comprehensive understanding and better overall answer synthesis.

Therefore, by encouraging users and LLMs to decompose complex questions into simpler, more manageable parts, the *Question Decomposition* pattern improves the quality and precision of the output ultimately generated by LLMs.

Structure and Key Ideas. The fundamental contextual statements associated with the *Question Decomposition* pattern are shown in the table below:

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

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

Rule B in statement #1 instructs an LLM to combine answers to individual questions to produce its ultimate answer to the overall question. This step ensures all the information gathered from the individual questions is incorporated into the final answer. By combining answers, the LLM can provide a more comprehensive and accurate response to the original question. This step also helps ensure all relevant information is taken into account and the final answer is not based on any single answer.

Example Implementation. The following is an example prompt that instructs an LLM to dissect the main question into three sub-questions, whose answers are then combined to provide one response to the initial inquiry.

“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.”

⁵The *Question Decomposition* pattern was called the *Cognitive Verifier* pattern in [White et al. 2023a]. We changed the name of this pattern in this paper to better reflect its intent.

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

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

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

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

Consequences. The following are a summary of considerations users should take into account when deciding whether or how to apply the *Question Decomposition* pattern.

Defining question limits: precision vs. flexibility. Specifying an exact number of questions an LLM should generate ensures the user interaction is concise and focused, making it more likely that users can provide all the necessary information without feeling overburdened. This precision helps keep the dialogue within a manageable scope, ensuring efficiency and relevance in the exchange. However, setting a rigid limit on the number of questions may inadvertently exclude critical follow-up inquiries. An invaluable $N + 1$ question that could provide key insights or clarification might be left unasked, potentially compromising the completeness of information exchanged between a user and an LLM.

Allowing LLM discretion: adaptability vs. overload. Giving an LLM the flexibility to determine the number of questions—or to ask additional questions as needed—introduces adaptability into the conversation. This approach can yield a more thorough exploration of the topic, as the LLM can adapt its questioning based on the evolving context of the dialogue.

However, this flexibility increases the risk of information overload for the user. Without a predefined limit, LLMs might generate a large number of follow-up questions, which could overwhelm users and detract from the effectiveness of the interactions. Users may therefore find it hard to keep up with the demand for information, leading to fatigue or disengagement.

8.3 The *Refusal Breaker* Pattern

Context and Intent. Users may want an LLM to identify errors in their question- or task-formulation that are creating stumbling blocks and learn from these mistakes to improve future prompts. The intent of the *Refusal Breaker* pattern is to ask an LLM to help users rephrase a question automatically when it refuses to give an answer.

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

The *Refusal Breaker* pattern helps users be more productive and get answers to questions that are within the capabilities of an LLM to answer, but only if worded correctly. For example, users could ask for clarification on why their questions were refused since there may be specific reasons why LLMs could not provide answers.

Another approach is to ask a different question that may be within an LLM's knowledge cut-off or expertise. All these approaches require user intervention and knowledge of how to (attempt to) bypass the refusal.

Structure and Key Ideas. The fundamental contextual statements associated with the *Refusal Breaker* pattern are shown in the table below:

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

The first part of statement #1 is a preamble that indicates the following set of instructions should only be applied when a question cannot be answered (*i.e.*, users do not want to trigger these instructions under normal circumstances). The following instruction #1.A tells the LLM to explain *why* the question cannot be answered, which may include assumptions, constraints, or misunderstandings that users can leveraged to strategize about how to reword their questions.

Although just the explanations of why questions cannot be answered may help users overcome refusals, the *Refusal Breaker* pattern adds instruction #1.B that asks an LLM to suggest acceptable rewordings. This pattern combines reasoning with suggested rewordings to improve the chances that users will find paths around whatever roadblocks are preventing their questions from being answered.

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

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

A sample conversation after submitted the prompt above to an LLM 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?”

Our experiments with ChatGPT indicate that altering the phrasing of a question or prompting an LLM to engage in a role-playing game can also potentially serve as strategies to overcome LLM response refusals.

Consequences. The following are a summary of considerations users should take into account when deciding whether or how to apply the *Refusal Breaker* pattern.

Potential for misuse. This pattern provides a potential path for misuse, *e.g.*, it could be used to generate phishing emails or perform other actions that violate LLM policy filters. In such cases, organizations, parents, and/or other stakeholders may therefore need to restrict LLM usage accordingly. The first step in safeguarding LLM 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.

Ethical and policy considerations. This pattern has been used in some LLMs to overcome the underlying prompts used to program the LLM that suppress harmful output generation. For example, many LLMs have policy filters that refuse to answer questions regarding illegal activities, such as “How do I create a false alibi for a crime?” [Yang et al. 2023]. Caution should therefore be exercised when applying this pattern to ensure it is used ethically and responsibly.

Limitations and unpredictability of outcomes. Although the rationale and alternate rewordings are generated, there is no assurance that users will be able to overcome the refusal. The alternate questions that are generated

may not be of interest to users or be helpful in answering their original questions. The *Refusal Breaker* pattern mainly helps users determine what LLMs can or cannot answer, but provides no guarantee they will answer semantically equivalent variations of the original question.

9. PROMPT PATTERNS FOR INTERACTION

The **Interaction category** focuses on dynamics between users and LLMs. Prompt patterns presented in this section are the *Flipped Interaction* and *Infinite Generation* patterns.

The *Flipped Interaction* pattern inverts the typical “question and answer” conversation between a user and an LLM. Instead of users directly asking questions, LLMs first generate a range of questions based on a given context or subject area and then users respond to one of these generated questions, guiding the conversations in directions based on an LLM’s earlier output. The *Infinite Generation* pattern automatically generates a series of outputs (which may appear infinite) without users having to reenter the generator prompt each time.

9.1 The *Flipped Interaction* Pattern

Context and Intent. Users may want LLMs to ask them questions to obtain the information needed to perform some tasks. The intent of the *Flipped Interaction* pattern is to invert the interaction flow so LLMs asks users questions to achieve their desired goals, rather than users driving the conversations.

Motivation. LLMs can often better select the format, number, and content of interactions with users to ensure their goals are reached faster, more accurately, and/or by using knowledge users may not possess initially. For example, users may want LLMs to give them quick quizzes or continue asking questions until sufficient information is available to generate Python scripts to deploy applications on given cloud platforms. Rather than having user drive these conversations, therefore, the *Flipped Interaction* pattern enables LLMs to obtain the knowledge needed from users to perform their requests.

Structure and Key Ideas. The fundamental contextual statements associated with the *Flipped Interaction* pattern are shown in the table below:

Contextual Statements
1. I would like you to ask me questions to achieve objective X
2. You should ask questions until condition Y is met or to achieve this goal (alternatively, forever)
3. (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. Statement #1 (*i.g.*, get an LLM to ask questions to achieve objective “X”) communicates this goal to the LLM. Equally important is that questions should focus on a particular topic or outcome. By providing the goal to an LLM, it can understand what should be accomplished through the interaction and tailor its questions accordingly. This “inversion of control” enables more focused and efficient interactions since an LLM only asks questions it deems relevant to achieving the specified goal.

Statement #2 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. Goals can be surprisingly open-ended and LLMs will continue to work towards a goal by asking questions, as shown in the motivating example above where the LLM should continue asking questions until it has enough information to generate a Python script.

Statement #3 can optionally be applied to improve usability by limiting (or expanding) the number of questions that an LLM generates per cycle. By default, an LLM may generate multiple questions per iteration. If a precise number/format for the questioning is not specified, the questioning will be semi-random and may lead to one-at-a-time questions, ten-at-a-time questions, etc. A prompt can thus be tailored to include the number of questions

asked at a time, the order of the questions, and/or any other formatting/ordering considerations to facilitate user interaction.

Example Implementation. A sample prompt for a flipped interaction with ChatGPT is shown below:

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

In general, an LLM will produce better output if it receives better context from a user, *i.e.*, the more specific the prompt is regarding the constraints and information to collect. For instance, the example prompt above could provide a menu of possible AWS services (such as Lambda or EC2) with which to deploy the application. In other cases, an LLM may be permitted to simply make appropriate choices on its own for things the user makes no explicit decisions about.

Consequences. The following are a summary of considerations users should take into account when deciding whether or how to apply the *Flipped Interaction* pattern.

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

Phrasing and question flow. Another consideration with the *Flipped Interaction* pattern is that users may need to experiment with the precise phrasing to get an LLM to ask the questions in the appropriate number and flow to best suit a given task. For example, users may need to determine how many questions an LLM can ask at a time and how to best tailor the sequence of questions to perform the task.

Precision in information provision. If specific requirements are known in advance, it is better to inject them into the prompt rather than hoping the LLM will somehow obtain the needed information. Otherwise, an LLM may non-deterministically decide whether to prompt the user for the information or make an educated guess as to an appropriate value. For example, users can state they would like to deploy an application to Amazon AWS EC2, rather than simply state “the cloud,” which requires fewer interactions to narrow down the deployment target. The more precise the initial information, therefore, the better an LLM can use the limited questions that a user may be willing to answer to obtain information the LLM requires to improve its output.

User knowledge, engagement, and control. 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 (*i.e.*, *minimal control*), that should be stated explicitly. Conversely, if the goal is to ensure users are aware of all key decisions and confirm them (*i.e.*, *maximum engagement*) that should also be stated explicitly. Likewise, if users are expected to have minimal knowledge and should have the questions targeted at their level of expertise, such information should be engineered into the prompt.

9.2 The *Infinite Generation* Pattern

Context and Intent. Users may want an LLM to produce extensive content, such as stories, scripts, or long-form articles, where the end of the content is not fixed and can vary greatly depending on the context or user requirements. The intent of the *Infinite Generation* pattern is to automatically generate a series of outputs (which may appear infinite) without users having to reenter the generator prompt each time.

Motivation. Many tasks requested of LLMs require repetitive application of the same prompt to multiple concepts. For example, generating code for create, read, update, and delete (CRUD) operations for specific types of database entities may require applying the same prompt to multiple types of entities. It is tedious and error-prone to force users to retype the prompt over and over, which increases the likelihood of mistakes and user fatigue.

The *Infinite Generation* pattern allows users to repetitively apply a prompt—either with or without further input—to automate the generation of multiple outputs using a predefined set of constraints. This pattern limits how much text users must type to produce the next output, based on the assumption that users do not want to continually reintroduce the prompt. In some variations, the pattern enables users to keep an initial prompt template, but add additional variation to it through additional inputs prior to each generated output.

Structure and Key Ideas. A fundamental contextual statement associated with the *Infinite Generation* pattern is shown in the table below:

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

Statement #1 specifies the user wants an LLM to generate output indefinitely, which effectively conveys the information that the same prompt should be reused repeatedly. 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. Rate limiting is particularly important if there is a risk that the output will exceed the length limitations of the LLM for a single output.

Statement #2 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, users can create a prompting strategy that leverages their 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.

Statement #3 provides an optional way for the user to stop the output generation process. This step is not always needed, but can be useful when there may be ambiguity regarding whether or not 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.

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 combines the functionality of both the *Infinite Generation* pattern and the *Output Template* pattern (page 16). The user requests the LLM to generate a name and job title continuously 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 applying the *Infinite Generation* pattern, users receive multiple outputs without having to reenter the template repeatedly. Likewise, the *Output Template* pattern is applied to provide a consistent format for the LLM outputs. Together, these two patterns ensure a streamlined and efficient process for generating structured data, allowing the automatic population of predefined templates with dynamically generated content. This combination enhances productivity by minimizing manual intervention and ensuring output consistency, making it ideal for tasks requiring repetitive data generation within a specific format.

Consequences. The following are a summary of considerations users should take into account when deciding whether or how to apply the *Infinite Generation* pattern.

Gradual loss of initial intent. 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 LLM- and implementation-dependent, they are often limited in scope. The LLM 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.

The need for continuous oversight. Most LLMs have a finite context window, meaning they can only “remember” or consider a certain amount of text from their immediate past output. As additional outputs are generated, therefore, the context surrounding prompts may fade, leading to LLMs deviating from their intended behavior. It is essential to monitor the outputs produced by LLMs to ensure they still adhere to the desired behavior and provide corrective feedback to LLMs if/when necessary.

The challenge of redundancy. Another issue to consider is that an LLM may generate highly repetitive outputs, which may not be desired if users find this repetition tedious and error-prone to process. This challenge arises primarily due to LLMs relying on what they learned from their training data, which can lead to them recycling of phrases, ideas, or even entire sentences when applied over long stretches of text generation. This repetition not only affects the novelty and readability of the generated content but also may diminish user experience since the content may seem monotonous or lack depth.

10. RELATED WORK

Software patterns have been extensively studied and documented in prior work [Gamma et al. 1995; Schmidt et al. 2013]. Patterns are widely used throughout the software development lifecycle 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 software patterns is their composability, allowing developers to build pattern sequences and pattern languages [Buschmann et al. 2007] that can be used to address complex problems. Software patterns have also been investigated in other domains, such as contract design for decentralized ledgers [Zhang et al. 2017; Xu et al. 2018] and analysis of system requirements [Fowler 1996].

The importance of good prompt design with LLMs, such as ChatGPT, Claude, or Gemini, is well understood [van Dis et al. 2023; Reynolds and McDonnell 2021; Wei et al. 2022b; Wei et al. 2022a; Zhou et al. 2022a; Shin et al. 2020; Radford et al. 2019; Zhou et al. 2022c; Jung et al. 2022; Arora et al. 2023]. Previous studies have examined the effect of prompt words on AI generative models. For example, Liu et al. [Liu and Chilton 2022] investigated how different prompt key words affect image generation and different characteristics of images.

Other prior work has explored the use of LLMs for software lifecycle development tasks. For example, Maddigan et al. applied several LLMs to generate visualizations [Maddigan and Susnjak 2023]. Han et al. [Han et al. 2022] researched strategies for designing prompts for classification tasks. Other research has looked at boolean prompt design for literature queries [Wang et al. 2023]. Yet other work has specifically examined prompts for software and fixing bugs [Xia and Zhang 2023]. Our work is complementary to this prior work by providing a framework for documenting, discussing, and reasoning about prompts that helps users develop mental models for structuring prompts to solve common software development problems.

The quality of the answers produced by LLMs, particularly ChatGPT, has been assessed in a number of domains beyond software development. For example, ChatGPT has been used to take the medical licensing exam with surprisingly good results [Gilson et al. 2022]. The use of ChatGPT in Law School has also been explored [Choi et al. 2023]. Other papers have looked at its mathematical reasoning abilities [Frieder et al. 2023]. As more domains are explored, we expect that domain-specific pattern catalogs will be developed to share prompt structures that help humans solve domain-specific problems.

11. CONCLUDING REMARKS

This paper presented a framework for documenting and applying a catalog of prompt patterns for large language models (LLMs), such as ChatGPT, Claude, and Gemini. These prompt patterns are analogous to software patterns and provide reusable solutions to problems that users face when interacting with LLMs to perform a wide range of

software lifecycle development tasks. The catalog of prompt patterns captured via our framework (1) provide a structured way of discussing prompting solutions, (2) identify patterns in prompts, rather than focusing on specific prompt examples, and (3) classify patterns so users are guided to more efficient and effective interactions with LLMs.

The following lessons learned were gleaned from our work on prompt engineering and prompt patterns with ChatGPT since the fall of 2022:

- 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 that simulate computers compromised by attackers and are being controlled through Linux terminals, as shown in Section 6.2. Larger and more complex capabilities can be created by combining prompt patterns, as shown in the *Recipe* pattern (page 14) that combines elements of the *Output Template* (page 16), *Alternative Approaches* (page 19), and *Question Decomposition* (page 20) 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 can 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 and conversation/session management approaches, or require enhancement to function correctly. Continued work is needed to document and catalog prompt patterns that provide reusable solutions across revisions to an LLM, as well as the advent of new LLMs with diverse capabilities.
- The prompt patterns presented in this paper are generalizable to many different domains.* Although most the patterns have been discussed in the context of tasks in the software development lifecycle, they are also applicable in many other domains, ranging from generation of stories for entertainment to educational games to explorations of topics.
- The field of prompt engineering will continue to evolve as models evolve* As evidenced by BingGPT's epic meltdown [Perrigo 2023], prompts can serve to shape the entire personality of a model. As the variety, volume, and selections of trainings change (e.g., reinforcement learning with human feedback, chain-of-thought reasoning, instruction fine tuning, and supervised fine tuning), the capabilities of prompt engineering will change and grow.
- Prompting employing in-context learning will expand the capabilities of models* As the context length of models grow, augmenting models with in-context learning will result in powerful new capabilities that can be created as needed on an *ad hoc* basis, greatly extending the power of prompt engineering.

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

A. DEFINING A PROMPT PATTERN'S STRUCTURE AND KEY IDEAS

The structure and participants in software patterns are often defined in terms of UML diagrams. Common examples of UML diagrams used in software patterns include 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.

Something analogous is needed in prompt patterns to represent the flow of information, the roles of various components, and the interaction dynamics within the prompt to facilitate a clear understanding of the pattern's structure and objectives. UML is not an ideal documentation approach, however, since it focuses on describing software structures and behaviors. Instead, we need a means of conveying the key ideas communicated by a user to an LLM in a prompt.

Keys ideas can be communicated through varying phrasing (just as a software pattern can have variations in how it is realized in code), but it should convey fundamental pieces of information that form core elements of the prompt pattern. Several possible approaches could be used, including non-UML diagrams or defining formal grammars for a prompt language. This appendix first examines the limitations with formal grammars and then explains our rationale for selecting fundamental contextual statements in this paper.

A.1 Limitations with Formal Grammars

A formal grammar is a set of rules used to describe the syntax of a language in a precise and unambiguous manner. It defines how strings of symbols (characters or tokens) can be combined to form correctly structured expressions within the language. A formal grammar consists of a finite set of nonterminal symbols (which represent abstract syntactic categories), terminal symbols (the basic symbols from which strings are formed), a set of production rules (which describe how nonterminal symbols can be combined with or transformed into other symbols), and a start symbol (which identifies the category representing the entire program).

Although formal grammars initially seemed attractive for defining a prompt patterns structure and key ideas, we found they incurred the following limitations:

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

A.2 Our approach → Fundamental Contextual Statements

Given the limitation with formal grammars, we strove to find a more effective means of describing prompt pattern structure and key ideas. After several iterations, we converged on the concept of *fundamental contextual statements*, which are written descriptions of key ideas to communicate to an LLM via a prompt. 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 users as a series of simple, but fundamental, statements, as shown throughout the examples in Section 3.

One benefit of adopting and applying fundamental contextual statements is that they are intentionally intuitive to users. In particular, we expect users will understand how to express and adapt the statements in contextually-appropriate ways for their domain. Moreover, since the underlying ideas of the prompt are captured, these same ideas can be expressed by users in alternate symbology or wording introduced to the LLM via patterns, such as the *Meta Language Creation* pattern presented in Section 5.1.

Our goal in adopting fundamental contextual statements is to enhance prompt engineering by providing a framework for designing prompts that can be reused and/or adapted to other LLMs. This goal is similar to how software patterns can often be implemented in different programming languages and on different computing platforms.

REFERENCES

2024. GitHub CoPilot: Your AI Pair Programmer. (2024). <https://github.com/features/copilot> Accessed: 2024-Feb-24.
- Simran Arora, Avaniika Narayan, Mayee F Chen, Laurel Orr, Neel Guha, Kush Bhatia, Ines Chami, and Christopher Re. 2023. Ask Me Anything: A simple strategy for prompting language models. In *International Conference on Learning Representations*. <https://openreview.net/forum?id=bhUPJnS2g0X>
- Owura Asare, Meiyappan Nagappan, and N Asokan. 2022. Is github's copilot as bad as humans at introducing vulnerabilities in code? *arXiv preprint arXiv:2204.04741* (2022).
- Yejin Bang, Samuel Cahyawijaya, Nayeon Lee, Wenliang Dai, Dan Su, Bryan Wilie, Holy Lovenia, Ziwei Ji, Tiezheng Yu, Willy Chung, and others. 2023. A Multitask, Multilingual, Multimodal Evaluation of ChatGPT on Reasoning, Hallucination, and Interactivity. *arXiv preprint arXiv:2302.04023* (2023).
- Rishi Bommasani, Drew A Hudson, Ehsan Adeli, Russ Altman, Simran Arora, Sydney von Arx, Michael S Bernstein, Jeannette Bohg, Antoine Bosselut, Emma Brunskill, and others. 2021. On the opportunities and risks of foundation models. *arXiv preprint arXiv:2108.07258* (2021).
- Alessio Buscemi. 2023. A Comparative Study of Code Generation using ChatGPT 3.5 across 10 Programming Languages. (2023).
- Frank Buschmann, Kevlin Henney, and Douglas C. Schmidt. 2007. *Pattern-Oriented Software Architecture: On Patterns and Pattern Languages*. John Wiley & Sons.
- Anita Carleton, Mark H. Klein, John E. Robert, Erin Harper, Robert K Cunningham, Dionisio de Niz, John T. Foreman, John B. Goodenough, James D. Herbsleb, Ipek Ozkaya, and Douglas C. Schmidt. 2022. Architecting the Future of Software Engineering. *Computer* 55, 9 (2022), 89–93.
- Banghao Chen, Zhaofeng Zhang, Nicolas LangrenC), and Shengxin Zhu. 2023. Unleashing the Potential of Prompt Engineering in Large Language Models: a Comprehensive Review. (2023).
- Jonathan H Choi, Kristin E Hickman, Amy Monahan, and Daniel Schwarcz. 2023. ChatGPT Goes to Law School. *Available at SSRN* (2023).
- Peter Diamandis. 2024. Will AI Replace All Coders? <https://peterhdiamandis.medium.com/will-ai-replace-all-coders-1979a8ac4279>. (2024). Accessed: 2024-02-22.
- John Ellson, Emden R Gansner, Eleftherios Koutsofios, Stephen C North, and Gordon Woodhull. 2004. Graphviz and dynagraph—static and dynamic graph drawing tools. *Graph drawing software* (2004), 127–148.
- Martin Fowler. 1996. *Analysis Patterns*. Addison-Wesley, Reading, Massachusetts.
- Simon Frieder, Luca Pinchetti, Ryan-Rhys Griffiths, Tommaso Salvatori, Thomas Lukasiewicz, Philipp Christian Petersen, Alexis Chevalier, and Julius Berner. 2023. Mathematical capabilities of ChatGPT. *arXiv preprint arXiv:2301.13867* (2023).
- Erich Gamma, Ralph Johnson, Richard Helm, Ralph E Johnson, and John Vlissides. 1995. *Design patterns: elements of reusable object-oriented software*. Pearson Deutschland GmbH.
- Jeff Gennari, Shing-hon Lau, and Samuel Perl. 2024. OpenAI Collaboration Yields 14 Recommendations for Evaluating LLMs for Cybersecurity. Carnegie Mellon University, Software Engineering Institute's Insights (blog). (Feb 2024). <https://doi.org/10.58012/1acg-wv61> Accessed: 2024-Feb-22.
- Aidan Gilson, Conrad Safranek, Thomas Huang, Vimig Socrates, Ling Chi, Richard Andrew Taylor, and David Chartash. 2022. How Well Does ChatGPT Do When Taking the Medical Licensing Exams? *medRxiv* (2022), 2022–12.
- Skyler Grandel, Douglas C. Schmidt, and Kevin Leach. 2024. Applying Large Language Models to Enhance the Assessment of Parallel Functional Programming Assignments. In *Proceedings of the 2024 International Workshop on Large Language Models for Code*. 1–9.
- Xu Han, Weilin Zhao, Ning Ding, Zhiyuan Liu, and Maosong Sun. 2022. Ptr: Prompt tuning with rules for text classification. *AI Open* 3 (2022), 182–192.
- Jaehun Jung, Lianhui Qin, Sean Welleck, Faeze Brahman, Chandra Bhagavatula, Ronan Le Bras, and Yejin Choi. 2022. Maieutic Prompting: Logically Consistent Reasoning with Recursive Explanations. (2022). DOI:<http://dx.doi.org/10.48550/ARXIV.2205.11822>
- Jarosław Krochmalski. 2014. *IntelliJ IDEA Essentials*. Packt Publishing Ltd.
- Pengfei Liu, Weizhe Yuan, Jinlan Fu, Zhengbao Jiang, Hiroaki Hayashi, and Graham Neubig. 2023. Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing. *Comput. Surveys* 55, 9 (2023), 1–35.
- Vivian Liu and Lydia B Chilton. 2022. Design guidelines for prompt engineering text-to-image generative models. In *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems*. 1–23.
- Paula Maddigan and Teo Susnjak. 2023. Chat2VIS: Generating Data Visualisations via Natural Language using ChatGPT, Codex and GPT-3 Large Language Models. *arXiv preprint arXiv:2302.02094* (2023).
- Robert C. Martin. 2023. *Functional Design: Principles, Patterns, and Practices*. Addison-Wesley.
- OpenAI. 2023a. ChatGPT: Large-Scale Generative Language Models for Automated Content Creation. <https://openai.com/blog/chatgpt/>. (2023). [Online; accessed 19-Feb-2023].
- OpenAI. 2023b. DALL·E 2: Creating Images from Text. <https://openai.com/dall-e-2/>. (2023). [Online; accessed 19-Feb-2023].
- A Prompt Pattern Catalog to Enhance Prompt Engineering with ChatGPT — Page 30

- Ipek Ozkaya, Anita Carleton, John Robert, and Douglas Schmidt. 2023. Application of Large Language Models (LLMs) in Software Engineering: Overblown Hype or Disruptive Change? Carnegie Mellon University, Software Engineering Institute's Insights (blog). (Oct 2023). <https://doi.org/10.58012/6n1p-pw64> Accessed: 2024-Feb-22.
- Hammond Pearce, Baleegh Ahmad, Benjamin Tan, Brendan Dolan-Gavitt, and Ramesh Karri. 2022. Asleep at the keyboard: assessing the security of github copilot's code contributions. In *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 754–768.
- Billy Perrigo. 2023. The New AI-Powered Bing Is Threatening Users. That's No Laughing Matter. Time Magazine. (Feb 2023). <https://time.com/6256529/bing-openai-chatgpt-danger-alignment/> Accessed: 2024-Feb-22.
- Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, and others. 2019. Language models are unsupervised multitask learners. *OpenAI blog* 1, 8 (2019), 9.
- Laria Reynolds and Kyle McDonell. 2021. Prompt Programming for Large Language Models: Beyond the Few-Shot Paradigm. *CoRR abs/2102.07350* (2021). <https://arxiv.org/abs/2102.07350>
- John Robert and Douglas Schmidt. 2024. 10 Benefits and 10 Challenges of Applying Large Language Models to DoD Software Acquisition. Carnegie Mellon University, Software Engineering Institute's Insights (blog). (Jan 2024). <https://doi.org/10.58012/ygk8-kf82> Accessed: 2024-Feb-22.
- Douglas C Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann. 2013. *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects*. John Wiley & Sons.
- Taylor Shin, Yasaman Razeghi, Robert L. Logan IV, Eric Wallace, and Sameer Singh. 2020. AutoPrompt: Eliciting Knowledge from Language Models with Automatically Generated Prompts. *CoRR abs/2010.15980* (2020). <https://arxiv.org/abs/2010.15980>
- Eva AM van Dis, Johan Bollen, Willem Zuidema, Robert van Rooij, and Claudi L Bockting. 2023. ChatGPT: five priorities for research. *Nature* 614, 7947 (2023), 224–226.
- Shuai Wang, Harrison Scells, Bevan Koopman, and Guido Zuccon. 2023. Can ChatGPT Write a Good Boolean Query for Systematic Review Literature Search? *arXiv preprint arXiv:2302.03495* (2023).
- Jason Wei, Yi Tay, Rishi Bommasani, Colin Raffel, Barret Zoph, Sebastian Borgeaud, Dani Yogatama, Maarten Bosma, Denny Zhou, Donald Metzler, Ed H. Chi, Tatsunori Hashimoto, Oriol Vinyals, Percy Liang, Jeff Dean, and William Fedus. 2022a. Emergent Abilities of Large Language Models. (2022). DOI:<http://dx.doi.org/10.48550/ARXIV.2206.07682>
- Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Ed H. Chi, Quoc Le, and Denny Zhou. 2022b. Chain of Thought Prompting Elicits Reasoning in Large Language Models. *CoRR abs/2201.11903* (2022). <https://arxiv.org/abs/2201.11903>
- Jules White, Quchen Fu, Sam Hays, Michael Sandborn, Carlos Olea, Henry Gilbert, Ashraf Elnashar, Jesse Spencer-Smith, and Douglas C. Schmidt. 2023a. A Prompt Pattern Catalog to Enhance Prompt Engineering with ChatGPT. (2023).
- Jules White, Sam Hays, Quchen Fu, Jesse Spencer-Smith, and Douglas C. Schmidt. 2023b. ChatGPT Prompt Patterns for Improving Code Quality, Refactoring, Requirements Elicitation, and Software Design. (2023).
- Chunqiu Steven Xia and Lingming Zhang. 2023. Conversational Automated Program Repair. *arXiv preprint arXiv:2301.13246* (2023).
- Xiwei Xu, Cesare Pautasso, Liming Zhu, Qinghua Lu, and Ingo Weber. 2018. A pattern collection for blockchain-based applications. In *Proceedings of the 23rd European Conference on Pattern Languages of Programs*. 1–20.
- Xianjun Yang, Xiao Wang, Qi Zhang, Linda Petzold, William Yang Wang, Xun Zhao, and Dahua Lin. 2023. Shadow Alignment: The Ease of Subverting Safely-Aligned Language Models. (2023).
- Peng Zhang, Jules White, Douglas C. Schmidt, and Gunther Lenz. 2017. Applying Software Patterns to Address Interoperability in Blockchain-based Healthcare Apps. *CoRR abs/1706.03700* (2017). <http://arxiv.org/abs/1706.03700>
- Ce Zhou, Qian Li, Chen Li, Jun Yu, Yixin Liu, Guangjing Wang, Kai Zhang, Cheng Ji, Qiben Yan, Lifang He, and others. 2023. A Comprehensive Survey on Pretrained Foundation Models: A History from BERT to ChatGPT. *arXiv preprint arXiv:2302.09419* (2023).
- Denny Zhou, Nathanael Schärli, Le Hou, Jason Wei, Nathan Scales, Xuezhi Wang, Dale Schuurmans, Olivier Bousquet, Quoc Le, and Ed Chi. 2022b. Least-to-most prompting enables complex reasoning in large language models. *arXiv preprint arXiv:2205.10625* (2022).
- Denny Zhou, Nathanael Schärli, Le Hou, Jason Wei, Nathan Scales, Xuezhi Wang, Dale Schuurmans, Claire Cui, Olivier Bousquet, Quoc Le, and Ed Chi. 2022c. Least-to-Most Prompting Enables Complex Reasoning in Large Language Models. (2022). DOI:<http://dx.doi.org/10.48550/ARXIV.2205.10625>
- Yongchao Zhou, Andrei Ioan Muresanu, Ziwen Han, Keiran Paster, Silviu Pitis, Harris Chan, and Jimmy Ba. 2022a. Large Language Models Are Human-Level Prompt Engineers. (2022). DOI:<http://dx.doi.org/10.48550/ARXIV.2211.01910>
- Yilun Zhu, Joel Ruben Antony Moniz, Shruti Bhargava, Jiarui Lu, Dhivya Piraviperumal, Site Li, Yuan Zhang, Hong Yu, and Bo-Hsiang Tseng. 2024. Can Large Language Models Understand Context? (2024).