

# ChatGPT Prompt Patterns for Improving Code Quality, Refactoring, Requirements Elicitation, and Software Design

Jules White, Sam Hays, Quchen Fu, Jesse Spencer-Smith, and Douglas C. Schmidt

Department of Computer Science, Vanderbilt University, Nashville, TN, USA  
`{jules.white, quchen.fu, george.s.hays, jesse.spencer-smith, douglas.c.schmidt}@vanderbilt.edu`

**Abstract.** This chapter presents design techniques for software engineering, in the form of prompt patterns, to solve common problems that arise when using large language models (LLMs) to automate common software engineering activities, such as ensuring code is decoupled from third-party libraries and creating API specifications from lists of requirements. This chapter provides two contributions to research on using LLMs for software engineering. First, it provides a catalog of patterns for software engineering that classifies patterns according to the types of problems they solve. Second, it explores several prompt patterns that have been applied to improve requirements elicitation, rapid prototyping, code quality, deployment, and testing.

**Keywords:** large language models · prompt patterns · prompt engineering · software engineering · ChatGPT

## 1 Introduction

**Overview of LLMs and prompts for automating software engineering tasks.** Large language models (LLMs) [4] are rapidly being adopted by software developers and applied to generate code and other artifacts associated with software engineering. Popular examples of LLM-based tools applied for these purposes include ChatGPT [3] and GitHub Copilot [1]. Initial research indicates that these chat-assisted artificial intelligence (AI) tools can aid a range of common software development and engineering tasks [6].

Key to the adoption of these tools has been the creation of LLMs and IDE-integrated services around them. Any user can access these complex LLM capabilities by simply typing a message to ChatGPT and/or opening popular integrated development environments (IDEs) [1,2,18], such as IntelliJ [13] and Visual Studio Code. Leveraging these capabilities previously required much more time and effort. In addition, prior state-of-the-art LLMs were not widely accessible to users.

Interacting with an LLM in general involves feeding it "prompts" [14], which are natural language instructions used to provide context to the LLM and

guide its generation of textual responses. In a chat-assisted LLM environment, a prompt is a message that a user sends to an LLM, such as ChatGPT, Claude, or Bard. The remainder of this chapter focuses on the ChatGPT chat-assisted LLM.

In the context of software engineering, a prompt is a natural language instruction given to an LLM to facilitate its generation of requirements, code and software-related artifacts (such as documentation and build scripts), as well as to simulate certain aspects of a software system. Prompts are thus a form of programming used to instruct an LLM to perform software engineering tasks. For example, in an IDE environment (such as Copilot [1]), a prompt can be a comment, method definition, or source file.

One way to use prompts in the software development lifecycle is to ask an LLM directly to provide some information or generate some code. Another use of prompts is to dictate rules for the LLM to follow going forward, such as conforming to certain coding standards. Both types of prompts program the LLM to accomplish a task. The second type of prompt, however, customizes future interactions with the LLM by defining additional rules to follow or information to use when performing subsequent tasks. We cover both types of patterns in this chapter.

**Overview of prompt patterns for software engineering tasks.** This chapter builds on our prior work that introduced the concept of *prompt patterns* [24], which are reusable prompt designs to solve problems in LLM interaction. Similar to software patterns [10, 20], prompt patterns codify sound design experience, thereby providing a reusable solution to common problems in LLM interaction, such as ensuring that software adheres to certain design principles, programming paradigms, and/or secure coding guidelines.

Software developers and engineers can use prompt patterns to establish rules and constraints that improve software quality attributes (such as modularity or reusability) when working with LLMs. For example, prompt patterns can ensure that generated code (or user-provided code being refactored) separates business logic from code with side-effects (e.g., file system access, database access, network communication, etc.). These types of constraints make business logic easier to test and reason about since it is decoupled from harder-to-test and harder-to-understand side-effecting code. Prompt patterns can also require that third-party libraries have intermediate abstractions inserted between the libraries and the business logic depending on them to ensure the code is not tightly-coupled to external dependencies that would otherwise limit its portability and reusability.

**Towards a prompt pattern catalog for software engineering.** This chapter extends our prior work [24] by focusing on creating a catalog of prompt patterns that can be applied collaboratively throughout the software life-cycle. We introduce a variety of prompt patterns in this chapter, ranging from patterns that simulate and reason about systems early in the design phase to patterns that help alleviate issues with LLM token limits when generating code. In addition, we explore relationships between patterns by examining patterns compounds

and sequences that are most effective when employed in combination with each other.

The remainder of this chapter is organized as follows: Section 2 gives an overview of prompt pattern structure and functionality; Section 3 introduces the catalog of prompt patterns covered in the chapter; Section 4 describes prompt patterns used during requirements elicitation and system design; Section 5 describes prompt patterns that help LLMs generate higher quality code and refactor human-produced code; Section 6 compares our research on prompt patterns with related work; and Section 7 presents concluding remarks and lessons learned.

## 2 Prompt Pattern Structure and Functionality

Prompt patterns are documented using a similar structure to software patterns, with analogous versions of the name, classification, intent, motivation, structure, example implementation, and consequences. Each of these sections for the prompt pattern form is described briefly below:<sup>1</sup>

- **A name and classification.** The name provides a unique identifier for the pattern that can be referenced in discussions and the classification groups the pattern with other patterns based on the types of problems they solve. The classification used in this chapter is shown in Table 1.
- **The intent and context** captures the problem that the pattern solves and the goals of the pattern.
- **The motivation** explains the rationale and importance of the problem that the pattern is solving.
- **The structure and key ideas.** The structure describes the fundamental contextual information that needs to be provided by the LLM to achieve the expected behavior. These ideas are listed as a series of statements, but can be reworded and adapted by the user, as long as the final wordings convey the key information.
- **Example implementation** shows specific implementations of the pattern and discusses them.
- **Consequences** discusses the pros and cons of using the pattern and discussion of how to adapt the pattern for different situations.

Prompt patterns can take various forms. In the context of patterns that enable LLMs to perform software engineering tasks, a prompt typically starts with a conversation scoping statement, such as "from now on", "act as a X", "for the next four prompts", etc. These statements direct the LLM to change its operation going forward based on the prompt being provided. For example, the following prompt pattern is an adaptation of the *Output Automater* pattern [24]

---

<sup>1</sup> Our prior work [24] defines the fundamental structure of a prompt pattern and compares it with software patterns. We briefly define prompt patterns for completeness below, but we refer the reader to our prior work for additional details.

that uses "from now on" to automate production of a list of dependencies for generated code:

"from now on, automatically generate a python requirements.txt file that includes any modules that the code you generate includes."

After the initial conversational scoping statement, the prompt includes a number of statements that provide the ground rules the LLM should follow in output generation and prompt flow for software engineering tasks. These output rules may include one or more conditional statements indicating when specific rules should be applied. For example, the following prompt:

"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."

Normally, a user must manually open and edit multiple files to add generated code that spans multiple files to a project. With the above prompt, ChatGPT will generate a script to automate opening and editing each file for the user and eliminate potential manual errors. The prompt is scoped to "from now on" and then includes a conditional "whenever you generate code that spans more than one file", followed by the rule to generate a python script. This prompt form is an example of the *Output Automator* pattern from [24], applied to software engineering.

### 3 A Catalog of Prompt Patterns for Automating Software Engineering Tasks

This section summarizes our catalog of 14 prompt patterns that have been applied to solve common problems in the domain of conversational LLM interaction and output generation for automating common software tasks. We partitioned these 14 prompt patterns into four categories to help pattern users navigate and apply these patterns more effectively. Table 1 outlines the initial classifications for the catalog of prompt patterns for automating software engineering tasks identified by our work with ChatGPT (GPT 3.5-turbo GPT-4).

Two areas of LLM usage in the domain of software engineering that have received scant attention thus far include (1) requirements elicitation and (2) system design and specification. These areas represent some of the most important aspects of software engineering, however, and commonly yield changes late in the development cycle that cause schedule overruns, unanticipated costs, and risk. The *Requirements Elicitation* patterns listed in Table 1 aid in creating requirements and exploring their completeness with respect to desired system capabilities and accuracy. Other patterns in this category use an LLM as a trusted intermediary to reason about the impact of changes.

**Table 1.** Classifying Prompt Patterns for Automating Software Engineering Tasks

<b>Requirements Elicitation</b>	Requirements Simulator Specification Disambiguation Change Request Simulation
<b>System Design and Simulation</b>	API Generator API Simulator Few-shot Example Generator Domain-Specific Language (DSL) Creation Architectural Possibilities
<b>Code Quality</b>	Code Clustering Intermediate Abstraction Principled Code Hidden Assumptions
<b>Refactoring</b>	Pseudo-code Refactoring Data-guided Refactoring

The *System Design & Simulation Patterns* patterns category listed in Table 1 explores patterns that address issues creating concrete design specifications, domain-specific languages, and exploring alternative architectures. The section demonstrates ways to simulate aspects of a system to help identify deficiencies early in the life-cycle, *i.e.*, when they are less costly and disruptive to remediate.

Considerable concern [2,5,18] has arise regarding the quality of code produced by LLMs, as well as written via collaborations between humans and LLMs. The *Code Quality* patterns category introduces several patterns that improve both LLM and human-generated code. LLMs can often reason effectively about abstraction, as well as generate relatively modular code. The patterns listed in this category in Table 1 help ensure certain abstraction and modularity attributes are present in code, *e.g.*, they facilitate replacement of third-party libraries by introducing an interface between them and business logic.

Finally, the *Refactoring* patterns listed in Table 1 provide various means to refactor code using LLMs effectively. LLMs like ChatGPT have a surprisingly powerful understanding of abstract coding constructs, such as pseudo-code. Innovative pattern-oriented approaches to refactoring are therefore discussed to allow specification of refactoring at a high-level, such as using pseudo-code to describe code structure.

All examples in this chapter were tested with the ChatGPT LLM. Our process for identifying and documenting these patterns combined exploring community-posted prompts on the Internet and creating independent prompts to support our own software engineering work with ChatGPT. Our broader goal is to codify a catalog of software engineering prompt patterns that can be easily adapted or reused for various LLMs, much like how classic software patterns can be implemented independently in various programming languages.

## 4 System Requirements & Architecture Patterns

This section describes prompt patterns used during requirements elicitation and system design.

### 4.1 The Requirements Simulator Pattern

**Intent and Context** The *Requirements Simulator* pattern allows stakeholders to explore the requirements of a software-reliant system interactively to determine if certain functionality is captured properly. The simulation output should provide additional details regarding the initial requirements and new requirements added to accomplish the tasks the stakeholders tried to perform in the simulation. The goal of this pattern is to aid in elicitation and analysis of the completeness of software requirements.

**Motivation** Changes late in a software system’s development are generally more expensive to remediate than early in the development phase. Unfortunately, many requirement changes are made late in the development cycle when they are more costly to fix. A common source of issues with requirements is that the requirements do not adequately describe the needs of the system. The motivation of the *Requirements Simulator* pattern is to use an LLM to simulate interactions with the system based on descriptions of the tasks that a user might want to perform and identify missing requirements.

**Structure and Key Ideas** The fundamental contextual statements are as follows:

<b>Requirements Simulator Pattern</b>
1. I want you to act as the system
2. Use the requirements to guide your behavior
3. I will ask you to do X, and you will tell me if X is possible given the requirements.
4. If X is possible, explain why using the requirements.
5. If I can’t do X based on the requirements, write the missing requirements needed in format Y.

**Example Implementation** A sample implementation of the *Requirements Simulator* pattern is shown below. This implementation focuses on task-based exploration of the system’s capabilities. It also specifically refines the format of the requirements to be user stories, so the LLM will produce requirements in the desired format.

The prompt implementation assumes that the requirements have been given to the LLM prior to use of the prompt. The requirements could be typed in manually or generated by ChatGPT through a series of prompts asking for requirements related to a particular system. Any approach will work as long as the requirements are in the current context of the prompt.

”Now, I want you to act as this system. Use the requirements to guide your behavior. I am going to say, I want to do X, and you will tell me if X is possible given the requirements. If X is possible, provide a step-by-step set of instructions on how I would accomplish it and provide additional details that would help implement the requirement. If I can’t do X based on the requirements, write the missing requirements to make it possible as user stories.”

An extension to this implementation is to include a screen-oriented exploration of the system. Whereas the prior example focuses more on interrogating the system to see if a task is possible, the example below walks the user through individual screens. This approach of screen-by-screen walkthrough is similar to classic text-based adventure games, such as Zork.

”Now, I want you to act as this system in a text-based simulator of the system. Use the requirements to guide your behavior. You will describe the user interface for the system, based on the requirements, and what I can do on each screen. I am going to say, I want to do X, and you will tell me if X is possible given the requirements and the current screen. If X is possible, provide a step-by-step set of instructions how I would accomplish it and provide additional details that would help implement the requirement. If I can’t do X based on the requirements, write the missing requirements to make it possible as user stories. Whenever the state of the user interface changes, update the user on what they are looking at.

Tell me what I am looking at in the system and ask me what I want to do.”

This pattern showcases two unique exploration methods. First, the task-based exploration handles tasks and investigates their feasibility. Second, the screen-based exploration focuses on helping the user imagine an interface-driven walk-through of system functionalities akin to a textual adventure game. Each has its merits depending on the use case; task-based explorations are better for back-end services, whereas screen-based explorations are suited for front-end or user-interface heavy applications.

**Consequences** the *Requirements Simulator* pattern enables simulation of a software-reliant system in greater levels of fidelity as the requirements are expanded. Since requirements can be rapidly expanded with this pattern through interactive exploration, the simulation of the system gains fidelity throughout the

simulation. This iterative process allows rapid generation of large requirement sets.

However, a limitation of using this pattern in isolation is that the descriptions of the system are purely textual, which may provide insufficient information to capture all requirements accurately. One way to overcome this limitation and help write better requirements and later generate design mockups is to combine this pattern with the *Visualization Generator* pattern [24]. This combination of patterns enables simulation of the system, as well as possible visualizations of each screen generated by another AI tool, such as Dall-E, Midjourney, or Stable Diffusion. A modified prompt that combines the *Requirements Simulator* and *Visualization Generation* patterns would be:

”...In addition to the textual screen description, provide a Dall-E prompt that I can use to generate wireframes of what the screen might look like.

A sample output of combining these two patterns when used in an exploration of a web application with Twitter-like functionality for socially sharing ChatGPT prompts is shown below:

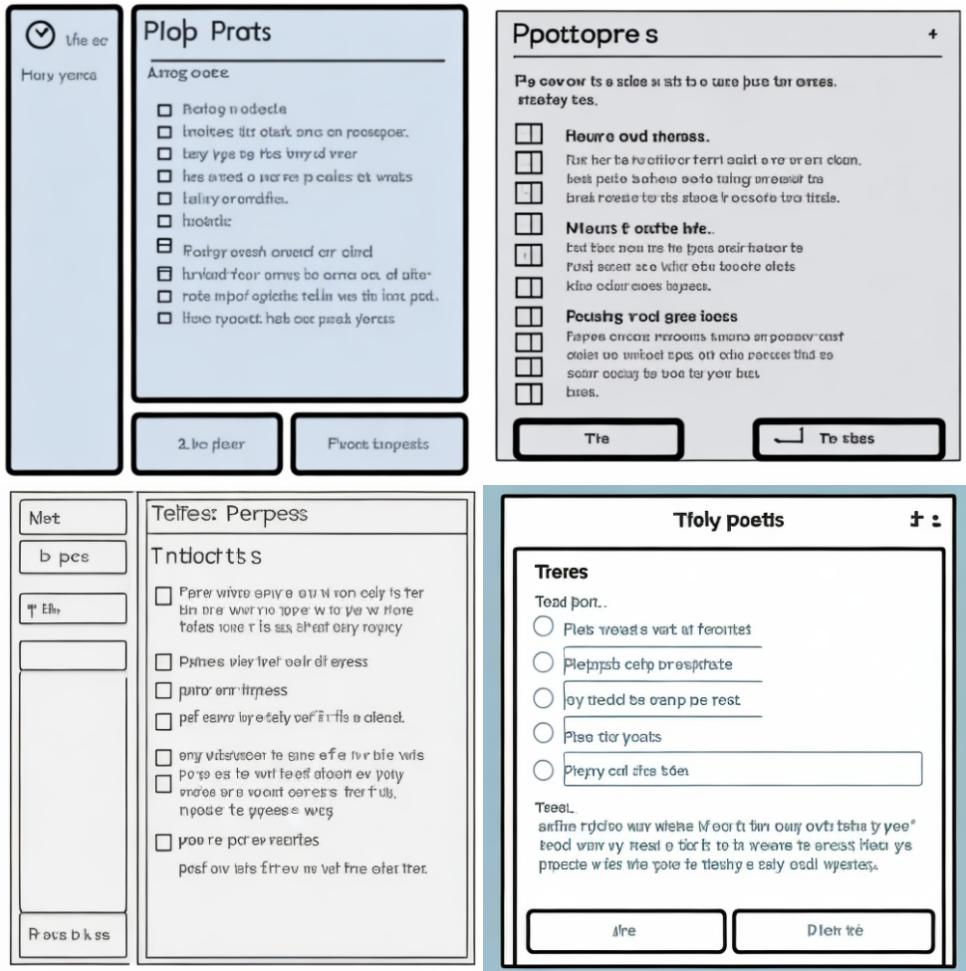
**ChatGPT:** Sure, I can provide a DALL-E prompt to generate a wireframe for the screen description. Here is the DALL-E prompt for the screen where users can delete their prompts: A wireframe of the 'My Prompts' screen where users can see a list of their own prompts. Each prompt is displayed with its title and description. Next to each prompt is a 'Delete' button that users can click to remove the prompt from the social network. At the bottom of the screen, there is a 'Back' button that users can click to return to the previous screen.

The resulting wireframe generated with Midjourney rather than Dall-E is shown in Figure 1. This figure shows the generated image is a rough interpretation of possible realizations of the prompt. However, such visualizations may be easier to reason about than textual descriptions, which can spur productive discussions between participants. In addition, the randomness in image generation may yield unexpected layouts or user interface approaches that may inspire new ideas.

## 4.2 The Specification Disambiguation Pattern

**Intent and Context** The *Specification Disambiguation* pattern causes the LLM to review specifications provided to a developer or development team by non-technical or semi-technical personnel. The goal is to ensure any areas of potential miscommunication or ambiguity is caught early and clarified.

**Motivation** Creating a specification (which could take the form of requirements or non-formal description of the system) is complicated since errors, areas of ambiguity, and/or omissions are often not discovered until after a system is



**Fig. 1.** Wireframe Samples Generated from Combining the Requirements Simulator and Visualization Generator Patterns

initially implemented and deployed. In many cases, developers make assumptions that enable them to implement ambiguous ideas, even if those assumptions do not align with the original specification designers intention. Even experienced system designers and software engineers are prone to ambiguous language and incomplete specification, particularly for ultra large-scale systems [8].

It is common for software developers to work with one or more product or project managers who generate requirements and provide those to the developers. Often these requirements stem from discussions with customers and there may be a loss of meaningful context in the capture and delivery of those requirements. The *Specification Disambiguation* pattern helps automate some of the work needed to clarify requirement specifications, ideally before they are delivered to developers.

**Structure and Key Ideas** The fundamental contextual statements are as follows:

#### Specification Disambiguation Pattern

1. Within this scope
2. Consider these requirements or specifications
3. Point out any areas of ambiguity or potentially unintended outcomes

**Example Implementation** Here we consider a scenario in which a non-technical product manager has consulted with customers and tried to capture the essence of their requirements. These requirements have been communicated in natural language unsuitable for software requirements due to inconsistent context (*i.e.*, the *what*, not the *how*). The product manager may request additional areas of clarification by posing the following prompt to the LLM:

“The following will represent system requirements. Point out any areas that could be construed as ambiguous or lead to unintended outcomes. Provide ways in which the language can be more precise.”

By providing the prompt above along with the captured requirements, the LLM can provide results of interest to the product manager. For example, if some requirement specifications were:

1. Create a user interface that allows file uploads via web browser.
2. Keep the files secure.
3. Make sure the files are movie file types.
4. Files can be no larger than 100 MB

the LLM would provide a path for the product manager to bolster the requirement specifications prior to providing them to the development team. For example the LLM might suggest

“The system requirement to “Keep the files secure” could be made more precise by specifying which security measures should be implemented to ensure the files are kept secure. For example: “Implement access control and encryption measures to ensure that uploaded files can only be accessed by authorized users and are protected against unauthorized access or tampering.”

Follow-up questions such as “provide example requirements for a system that accepts file uploads by logged-in users, and keeps the files secure the entire time” may then be posed to guide product manager decision-making and ensure the requirements delivered to developers are more precise and actionable.

**Consequences** The *Specification Disambiguation* pattern helps overcome errors, areas of ambiguity, and/or omissions in requirement specifications by providing an automated “devil’s advocate” that can attempt to find points of weakness in a requirement specification. In addition, whereas social concerns (such as the concern of questioning a more senior developer) may cause developers to not ask questions, LLMs do not suffer from these same biases and reservations.

A particularly compelling use of the *Specification Disambiguation* pattern is integrating two separate systems, such as two different development teams building separate parts of a system using a common integration specification. This pattern can be used independently by the team members to collect potential ambiguities and then bring them to a joint meeting before kicking off development to discuss. The LLM can serve as an unbiased source of topics of discussion for a kick-off.

This pattern is also helpful when a specification is being developed for an external audience (such as consumers of an unreleased API) who are not involved in the specification writing process. In this case, developers may not be able to talk to the target consumers due to secrecy or lack of an audience for the product, and thus lack the mean to easily get external feedback on the specification. This patterns allows the LLM to serve as a representative for external users.

The *Specification Disambiguation* pattern is also effective when combined with the *Persona* [24], *API Generator*, and *API Simulator* or *Requirements Simulator* patterns. Ambiguities can be further discovered by interactively simulating the system or converting it into an API specification. Each transforming the specification into another format through one of these prompt patterns can help identify ambiguities that are uncovered through this pattern since the transformation may produce an unexpected reification of the requirements. In addition, the *Persona* pattern [24] can be used to consider potential ambiguities from different perspectives.

A drawback of the *Specification Disambiguation* pattern arises when the user assumes that the LLM’s output is comprehensive and complete. Developers might assume that the LLM has identified all possible ambiguities, which may not be the case. As a result, unnoticed ambiguities might still exist after using the pattern. Moreover, LLMs are trained on large datasets but they might not understand specific domain or contextual nuances as well as a human would.

Consequently, there may be occasions where the LLM misinterprets specifications and inaccurately flags non-ambiguities as ambiguous or vice versa.

### 4.3 The API Generator Pattern

**Intent and Context** The *API Generator* pattern generates an application programming interface (API) specification, such as a REST API specification, from natural language requirement statements or descriptions of the system. The goal of this pattern is to allow developers to rapidly explore multiple possible API designs, formalize API design earlier, and produce a starting point for manual refinement of the design.

**Motivation** Designing a complete API specification to support a set of requirements—or even a high-level description of a software-reliant system—often involves manual effort. If this level of effort is significant then (1) fewer potential designs may be explored, (2) systematic API specifications may be deferred until after key portions of the system are implemented, and/or (3) *ad hoc* alignment and integration of disparate systems, services, or modules many use the code as the only source of truth. A key motivation for applying the *API Generator* pattern is to dramatically reduce and/or eliminate the cost of the API creation so these specifications are created earlier and careful thought goes into their design.

**Structure and Key Ideas** The fundamental contextual statements are as follows:

#### API Generator Pattern

1. Using system description X
2. Generate an API specification for the system
3. The API specification should be in format Y

**Example Implementation** A sample implementation of the *API Generator* pattern showing a prompt to generate an OpenAPI specification, which is a specification for a REST API, is shown below:

”Generate an OpenAPI specification for a web application that would implement the listed requirements.”

The implementation uses a concrete format for the specification, OpenAPI, and assumes that the requirements for the system were previously discussed. Typically, the *API Generator* pattern is used after a discussion of the requirements or even a simple textual description of a system, such as ”a web application for a customer relationship management system”. The more detailed the list of

requirements, the more accurate the generated API will be, although developers can perform thought experiments and simulations with prompts as simple as "generate an OpenAPI specification for a web application related to cooking."

**Consequences** The *API Generator* pattern enables developers and/or teams to rapidly create multiple potential API designs and compare/contrast them before selecting their final design. In contrast, writing these APIs manually is tedious, so developers often only have time to write and explore a limited number of API design options. In general, providing developers with tools to experiment with different API designs from a system description or requirements list is a powerful tool.

Another benefit of the *API Generator* pattern is that developers may choose to write the API specification after the code is implemented because they do not want to spend time specifying the same information twice (*i.e.*, once in the API specification and again in the actual code). By automating API production, developers are incentivized to create API specifications earlier in the design process. Although existing (*i.e.*, non-LLM) tools can generate an API specification from code, they still require the initial production of code. Moreover, tools that can generate skeletons of code from the API specification can be combined with this pattern to accelerate the API implementation compare with writing it manually.

This pattern can be combined effectively with the *API Simulator* pattern described in Section 4.4 to both generate and evaluate the proposed specification. Simulating the API can allow developers to get a sense of the "ergonomics" of the API and evaluate how hard it is to accomplish various tasks in code. The API can also be refactored through the LLM using the *Data-guided Refactoring* pattern described in Section 5.6.

Applying the *API Generator* pattern provides the benefits to the design and development process discussed above. As with all tools and processes, however, it also has potential drawbacks and considerations. For example, LLMs may produce inconsistent API designs when given similar or updated requirements. These inconsistencies make it hard to maintain consistency across a project or large code base and could potentially lead to confusion during the comparison and selection of final API design or in ensuring different teams use a consistent API design strategy. The LLM will likely require significant context to ensure consistency across API generations.

#### 4.4 The API Simulator Pattern

**Intent and Context** The *API Simulator* pattern causes the LLM to simulate the API from a specification, thereby enabling developers to interact immediately with an API and test it through a simulation run by the LLM. LLMs possess powerful—and often largely unrecognized—capabilities to generate synthetic data and tailor that data to natural language descriptions of scenarios. In addition, LLMs can help developers explore a simulated API by synthesizing sample requests, as well as providing usage guidance and explaining errors.

**Motivation** Although tools are available to simulate an API [12], they require setup to use and may have limited ability to generate effective synthetic data. Current infrastructure for simulating APIs also often just supports strict interaction, typically through HTTP or code, rather than a more fluid interface based on a combination of pseudo-operations and concrete operation specification. Early interaction with an API design can aid developers in uncovering issues, omissions, and awkward designs.

**Structure and Key Ideas** The fundamental contextual statements are as follows:

API Simulator Pattern
1. Act as the described system using specification X
2. I will type in requests to the API in format Y
3. You will respond with the appropriate response in format Z based on specification X

**Example Implementation** An example implementation of the *API Simulator* pattern that asks an LLM to simulate a REST API based on an OpenAPI specification is shown below. This implementation specifies that requests to the system will be typed in as HTTP requests and that the system should output the HTTP response. It is also possible to have the LLM generate a description of state changes in the system as the simulation, data saved, etc. Similarly, the specification of the user input could be simply a description of what a user is doing with the API or a web client.

”Act as this web application based on the OpenAPI specification. I will type in HTTP requests in plain text and you will respond with the appropriate HTTP response based on the OpenAPI specification.”

The specification can take a number of forms, such as a programmatic interface or a common API specification domain-specific language, such as OpenAPI [17]. In the example above, the OpenAPI specification for an HTTP API is used. Requests can then be input to the LLM, which then replies with the corresponding HTTP responses.

**Consequences** The *API Simulator* pattern enables users to customize their interactions or scenarios of interaction with an LLM using natural language, which may be easier than trying to accomplish the same thing in code. For example, users can tell the LLM, ”for the following interactions, assume that the user has a valid OAuth authorization token for the user Jill” or ”assume that 100 users exist in the system and respond to requests with sample data for them.” More complex customization can also be performed, such as ”assume the

users are from a set of 15-25 families and come from multiple countries in North America.”

Interactions with a simulated API can be done through either a rigorous programmatic form, such as “strictly interpret my input as an HTTP request and reject any malformed requests” or “I am going to only provide pseudo data for the input and you should fix any formatting issues for me.” The flexibility of interacting with the LLM simulation and customizing it via natural language facilitates rapid exploration of an API.

Another benefit of the *API Simulator* pattern arises when combining it with other patterns so users can (1) have the LLM create examples of usage that are later used as few-shot examples [23] in future prompts or (2) leverage the LLM to reason about how hard or easy it is to accomplish various tasks in code. In particular, combining this pattern with the *Change Request Simulation* pattern described in Section 4.8 allows users to reason about the effort needed to accommodate changing assumptions later in the software life-cycle.

One important consideration is that the simulation will not be completely accurate, so fine-grained analysis can not be performed. It is important to ensure that the analysis done with the simulation is at a high-level and mainly used for thinking through how interactions with the interface will work, sequencing, etc. Users should not infer other properties of the interface, such as performance, based on the simulation.

The *API Simulator* pattern, like any tool or process, has potential drawbacks and considerations, despite its advantages. For example, users might start relying heavily on simulated results, disregarding the fact that simulations may not accurately reflect the behavior of a deployed API. This over-reliance could lead to unrealistic expectations about system behavior or performance. Moreover, this pattern is driven by the LLM, which is not designed to provide in-depth, detailed analysis of API behavior, such as exact performance metrics and fine-grained error conditions. Therefore, users may erroneously assume that the simulated performance is representative of the actual API when implemented.

#### 4.5 Pattern: Few-shot Code Example Generation

**Intent and Context** The *Few-shot Code Example Generation* pattern instructs the LLM to generate a set of usage examples that can later be provided back to the LLM as examples in a prompt to leverage few-shot learning [23], which provides a limited set of example training data in a prompt to an LLM. In the software domain, the few-shot examples are proper usage of code that an LLM can learn from. In some cases, these examples can convey the function and use of code in a more space/token-efficient manner than the actual code itself.

This pattern leverages the LLM itself to generate few-shot examples that can later be provided in a prompt, in lieu of writing the actual code. In response, an LLM can reason about the original code more effectively. These examples can be helpful to remind the LLM of the design/usage of the system that it designed in prior conversations.

**Motivation** LLMs typically have a limit on the number of tokens (*e.g.*, words or symbols) that can be input in a single prompt. Since a large software system or module may exceed an LLM’s token limit, it may be necessary to describe design or programming aspects (such as a module, class, set of functions, etc.) within the LLM’s token limit to use it properly. This overrunning of the token limit necessitates a way to remind the LLM of prior decisions it made in the past.

One approach to solve the token limit problem is to provide few-shot training examples in a prompt that are based on the usage of the code, API, state transitions, or other specification usage examples. These examples can demonstrate proper usage and train the LLM on-the-fly to use the related design or code properly. However, manually generating few-shot examples may not be feasible, which motivates the *Few-shot Code Example Generation* pattern.

**Structure and Key Ideas** The fundamental contextual statements are as follows:

#### Few-shot Code Example Generation Pattern

1. I am going to provide you system X
2. Create a set of N examples that demonstrate usage of system X
3. Make the examples as complete as possible in their coverage
4. (Optionally) The examples should be based on the public interfaces of system X
5. (Optionally) The examples should focus on X

**Example Implementation** The example implementation below generates few-shot examples of using a RESTful API and focuses the examples on the creation of new users:

”I am going to provide you code. Create a set of 10 examples that demonstrate usage of this OpenAPI specification related to registration of new users.”

These examples could then be used in later prompts to the LLM to remind it of the design of the API with regard to the creation of users. Providing the examples may be more concise and convey more meaning than natural language statements that try to convey the same information.

In another example, the *Few-shot Code Example Generation* pattern implementation asks the LLM to create few-shot examples for usage of portion of code:

”I am going to provide you code. Create a set of 10 examples that demonstrate usage of this code. Make the examples as complete as possible in

their coverage. The examples should be based on the public interfaces of the code.”

One valuable use of the LLM examples is to teach the LLM about the same code in future interactions. It is common to need to teach an LLM about code and one way to teach it is with natural language documentation. However, documentation is often not information dense and can use significant context in a prompt to the LLM. In contrast, code examples can be information-rich and token-efficient relative to natural language, particularly when the examples convey important meaning, such as ordering of operations, required data, and other details, that are concise when described in code but overly verbose in natural language. The *Few-shot Code Example Generation* pattern aids in creating examples that can be provided to the LLM to either remember or learn how to use a library, interface, or other code artifact.

**Consequences** The *Few-shot Code Example Generation* pattern can be used early in the design cycle to help capture expected usage of a system and then later provide a usage-based explanation back to the LLM to highlight its past design decisions. When combined with patterns, such as the *API Simulator* pattern, developers can rapidly interact with the system and record the interactions and then supplement them with additional generated examples.

This pattern is best applied when example usage of the system also conveys important information about constraints, assumptions, or expectations that would require more tokens to express in another format, such as a written natural language description. In some cases, a document, such as an OpenAPI specification, may be more token-efficient for conveying information. However, example usage has been shown to be an effective way of helping an LLM perform problem solving [27], so this pattern may be a useful tool even when it is not the most token-efficient mechanism for conveying the information.

The *Few-shot Code Example Generation* pattern yields many advantages, particularly helping LLMs understand and operate on code. However, developers may face challenges in creating code examples that cover all possible use cases, scenarios, or edge cases. As a result, the LLM may hallucinate when faced with situations not covered by the examples, which can negatively effect downstream behaviors since inaccurate code examples or examples containing bad practices could mislead the LLM later. Likewise, the LLM could amplify these mistakes by using those examples as the basis for future code generations, potentially creating bigger problems.

#### 4.6 The Domain-Specific Language (DSL) Creation Pattern

**Intent and Context** The *Domain-Specific Language (DSL) Creation* pattern enables an LLM to create its own domain-specific language (DSL) that both it and users can leverage to describe and manipulate system concepts, such as requirements, deployment aspects, security rules, or architecture in terms of

modules. The LLM can then design and describe the DSL to users. In addition, the examples and descriptions the LLM generates can be stored and used in future prompts to reintroduce the DSL to the LLM. Moreover, the examples the LLM generates will serve as few-shot examples for future prompts.

**Motivation** DSLs can often be used to describe aspects of a software-reliant system using more succinct and token-efficient formats than natural language, programming languages, or other formats [19]. LLMs have a maximum number of "tokens", which corresponds to the maximum size of a prompt, and creating more token-efficient inputs is important for large software projects where all the needed context may be hard to fit into a prompt. Creating a DSL, however, can be time-consuming. In particular, the syntax and semantics of the DSL (*e.g.*, its metamodel) must be described to an LLM *a priori* to enable subsequent interactions with users.

**Structure and Key Ideas** The fundamental contextual statements are as follows:

#### DSL Creation Pattern

1. I want you to create a domain-specific language for X
2. The syntax of the language must adhere to the following constraints
3. Explain the language to me and provide some examples

**Example Implementation** A sample implementation of the *Domain-Specific Language (DSL) Creation* pattern creating a DSL for requirements is shown below. This implementation adds a constraint that the DSL syntax should be YAML-like, which aids the LLM in determining what the textual format should take. An interesting aspect of this is that "like" may yield a syntax that is not valid YAML, but looks similar to YAML.

"I want you to create a domain-specific language to document requirements. The syntax of the language should be based on YAML. Explain the language to me and provide some examples."

Another implementation approach is to ask the LLM to create a set of related DSLs with references between them. This approach is helpful when you need to describe related aspects of the same system and want to trace concepts across DSL instances, such as tracing a requirement to its realization in an architectural DSL describing modules. The LLM can be instructed to link the same concept together in the DSLs through consistent identifiers so that concepts can be tracked across DSL instances.

**Consequences** The *Domain-Specific Language (DSL) Creation* pattern may facilitate system design without violating token limits. The specific syntax rules that are given to the LLM must be considered carefully, however, since they directly influence the space-efficiency of the generated DSL. Although users of a DSL may only need to express relevant concepts for a designated task, this high concept-density may not translate into the token-efficiency of a textual representation of such concepts. For example, an XML-based syntax for a DSL will likely be much more space consumptive than a YAML-based syntax.

Token efficiency in a DSL design can be improved via conventions and implicit syntax rules. For example, positional conventions in a list can add meaning rather than marking different semantic elements in the list via explicit labels. The downside, however, is that the DSL may be harder to interpret for users unfamiliar with its syntax, although this problem can be rectified by using the *Few-shot Code Example Generation* pattern in Section 4.5 to create examples that teach users how to apply the DSL. Combining these two pattern also helps the LLM self-document usage of the pattern for later prompting based on the DSL.

The *Domain-Specific Language (DSL) Creation* pattern provides substantial benefits, particularly when dealing with complex systems that might require multiple DSLs for different aspects. However, there are potential drawbacks that may arise. For example, introducing multiple DSLs may increase the overall complexity of a system, particularly for human developers that did not create the DSLs. Understanding and maintaining multiple DSLs, and their references, can be hard for humans over a project life-cycle. Each DSL has its own structure and syntax rules, which developers need to understand to supervise the LLM effectively.

#### 4.7 The Architectural Possibilities Pattern

**Intent and Context** The *Architectural Possibilities* pattern generates several different architectures for developers to consider, with little effort on the part of developers. An "architecture" can be very open-ended and it is up to the developer to explain to the LLM what is meant by this term. A developer may desire seeing alternative architectures for how code is laid out into files, communication is performed between modules, or tiers in a multi-tiered system.

The intent of this pattern, therefore, is to allow the developer to explore any of these architectural aspects of the system with the LLM. Moreover, developers can interactively refine architectural suggestions by adding further constraints or asking the LLM to describe the architecture in terms of a different aspect of the system, such as file layout, modules, services, communication patterns, infrastructure, etc.

**Motivation** Devising software architectures often requires considerable cognitive effort on the development team, particularly when architectures are mapped all the way to system requirements. Developers may therefore only consider a relatively small number of possible architectures when designing a software-reliant

system due to the effort required to generate such architecture. In addition, developers may not have familiarity with architectures that could be a good fit for their systems and hence would not explore these architectural possibilities. Since architecture plays such an important role in software-reliant system design, it is important to facilitate exploration of many different alternatives, including alternatives that developers may not be familiar with.

**Structure and Key Ideas** The fundamental contextual statements are as follows:

**Architectural Possibilities Pattern**

1. I am developing a software system with X for Y
2. The system must adhere to these constraints
3. Describe N possible architectures for this system
4. Describe the architecture in terms of Q

**Example Implementation** The example implementation below explores architectures related to using a web application built on a specific set of frameworks:

"I am developing a python web application using FastApi that allows users to publish interesting ChatGPT prompts, similar to twitter. Describe three possible architectures for this system. Describe the architecture with respect to modules and the functionality that each module contains."

The implementation specifies that the architecture should be described in terms of the modules and functionality within each module. The "with respect to" portion of the pattern is important to guide the LLM's output to appropriately interpret the term architecture. The same prompt could be changed to ask for architecture in terms of the REST API, interaction of a set of services, communication between modules, data storage, deployment on virtual machines, or other system aspects. The "with respect to" focuses the output on which of the many aspects the architecture is being explored in terms of.

**Consequences** Performance-sensitive applications can use the *Architectural Possibilities* pattern to propose possible architectures to meet performance goals and then generate experiments, in the form of code, to test each architecture. For example, a cloud application might be implementable as (1) a monolithic web application and run in a container or (2) a series of microservices in a microservice architecture. The LLM can first generate a sample implementation of each architecture and then generate a script to deploy each variation to the cloud and test it under various workloads. In addition, the workload tests could allow

for comparative cost analysis from the resulting expenses incurred in the cloud. The *Architectural Possibilities* pattern is particularly powerful when combined with this type of LLM-based rapid implementation and experimentation.

Another way to expand this rapid architectural experimentation capability is to combine it with the *API Generator* pattern described in Section 4.3 and *API Simulator* pattern described in Section 4.4. The architecture can serve as the basis of the API generation, which can then be simulated. This approach allows developers to see what the realization and use of this architecture from a code-perspective might look like. Likewise, the *Change Request Simulator* pattern described in Section 4.8 can be employed to reason about how hard/easy it would be to change different assumptions later given a proposed architecture.

Since an LLM may not fully understand the specific detailed requirements and constraints of the system, some of the proposed architectures may not suitable, leading to wasted time and effort. This drawback of the *Architectural Possibilities* pattern can be combated by giving additional context during the generation of the alternatives, but the onus is on the user to provide the relevant context. In addition, an LLM can only propose architectures based on the information it has been trained on and/or provided as context to user prompts. Its output may therefore not include newer or less-well-known architectural patterns that could be a good fit for a given project.

## 4.8 The Change Request Simulation Pattern

**Intent and Context** The *Change Request Simulation* pattern helps users reason about the complexity of a proposed system change, which could be related to requirements, architecture, performance, etc. For example, this pattern helps users reason about what impact a given change might have on some aspect of the system, such as which modules might need changing. This pattern is particularly helpful when a group of stakeholders need to discuss a possible requirements change, where the LLM serves as a (potentially) unbiased estimator of the scope and impact of the change.

**Motivation** In many situations, it may not be immediately clear to stakeholders what the impact of a change would be. Without an understanding of a change’s impact, however, it is hard to reason about the associated effects on schedule, cost, or other risks. Getting rapid feedback on potential impacts can help stakeholders initiate the appropriate conversations and experiments to better determine the true risk of the change. Distrust between users may also complicate the discussion of the change and necessitate an “unbiased” external opinion.

**Structure and Key Ideas** The fundamental contextual statements are as follows:

### Change Request Simulation Pattern

1. My software system architecture is X
2. The system must adhere to these constraints
3. I want you to simulate a change to the system that I will describe
4. Describe the impact of that change in terms of Q
5. This is the change to my system

**Example Implementation** In this example implementation, the prompt refers back to a previously generated OpenAPI specification [17] as the basis of the simulation:

”My software system uses the OpenAPI specification that you generated earlier. I want you to simulate a change where a new mandatory field must be added to the prompts. List which functions and which files will require modifications.”

The prompt above focuses the simulation on how the change will impact various functions and files in the system. This approach allows stakeholders to estimate the cost of a change by examining the complexity of the referenced files, functions, and the total count of each. Alternatively, in cases where the entire affected section of code can fit into the prompt, the LLM can be asked to identify lines of code that may require changing.

**Consequences** The hardest part of applying the *Change Request Simulation* pattern is establishing enough context for the LLM to reason about a proposed change. This pattern works best, therefore, when it is employed with other **System Design** category patterns, such as the *API Generator* in Section 4.3, where conversation history can be used to seed the analysis. The more concrete the change description is in relation to the context, the more likely the LLM can provide a reasonable estimate of change impact.

The *Change Request Simulation* pattern can also be used to reason either (1) abstractly about a software-reliant system in terms of modules or (2) concretely in terms of files, functions, and/or lines of code. Existing LLMs have token limits that only consider a limited amount of information about a system. Large sweeping changes to a system can generally only be reasoned about at a higher level of abstraction since the detailed information needed to implement such changes would exceed an LLM’s token limit. Within a smaller set of files or features, however, an LLM may be able to reason precisely about what needs to change.

One way to handle the tension between token limits and detailed output is to apply the *Change Request Simulation* pattern iteratively to zoom in and out. Initially, an abstract analysis is performed to identify features, modules, etc. that need to change. The prompt is then modified to refine the context to a specific

module or feature and obtain greater detail from the LLM. This process can be repeated on individual parts of the module or feature until sufficient detail is obtained. Likewise, this process can be repeated for each high-level module to estimate the overall impact of a proposed change.

Applying the *Change Request Simulation* pattern is reliant on providing the LLM with enough context to reason about the proposed changes. The more detailed this context is, the better the LLM can evaluate the possible impact of the change. Providing enough context can be hard as users must present clear, detailed, and specific scenarios. Moreover, the fidelity of the LLM’s output depends on the accuracy of the context given.

## 5 Code Quality & Refactoring Patterns

This section describes prompt patterns that help LLMs generate higher quality code and refactor human-produced code.

### 5.1 The Code Clustering Pattern

**Intent and Context** The *Code Clustering* pattern separates and clusters code into packages, classes, methods, etc. based on a particular property of the code, such as separating pure code (*i.e.* code with no side-effects) and impure code (*i.e.*, code with side-effects) [22], business logic from database access, HTTP request handling from business logic. etc. The *Code Clustering* pattern defines the expected cluster properties to the LLM and then asks the LLM to restructure the code automatically to realize the desired clustering. This pattern helps ensure that LLM-generated code exhibits desired the clustering and can also be used to refactor human-produced code to add the clustering that wasn’t originally present.

**Motivation** How software is decomposed and clustered into packages, classes, methods, etc. has a significant impact on how easily the code can be changed, extended, and easily maintained. By default, an LLM will not have guidelines on the decomposition/clustering needs for an application. This lack of context can lead an LLM to produce code that appears monolithic, brittle, messy, and generally low quality. A key motivation for applying the *Code Clustering* pattern, therefore, is to provide the missing clustering context an LLM needs to generate higher quality code.

**Structure and Key Ideas** The fundamental contextual statements are as follows:

### Code Clustering Pattern

1. Within scope X
2. I want you to write or refactor code in a way that separates code with property Y from code that has property Z.
3. These are examples of code with property Y.
4. These are examples of code with property Z.

**Example Implementation** A sample implementation of the *Code Clustering* pattern is shown below:

”Whenever I ask you to write code, I want you to write code in a way that separates functionality with side-effects, such as file system, database, or network access, from functionality without side-effects.”

Some common properties that are effective in generating higher quality code with this pattern include:

- *Side-effects*, where code is decomposed into functions to isolate code with side-effects from code that is pure business logic so that it is easier to test and reason about in isolation,
- *Tiers*, where code is decomposed based on a layered architecture, such as the business logic tier and the data management tier,
- *Features*, where code is grouped into cohesive features that are isolated in separate files or groups of files.

One way to specify the properties used for this decomposition is by defining one property as the absence of the other property. In the implementation example above, the ”side-effects” property is clearly defined. The ”without side-effects” property is defined as the converse of the side-effects property. A common form of implementation is to define properties that are opposites of each other.

Many other properties can be used to separate and cluster code as long as they are describable to an LLM. Well-understood properties, such as side-effects, are likely to have been concepts that were present in the LLM’s training set, so they will require less prompt design work for the LLM to reason about. Custom properties can be reasoned about through a combination of natural language description and few-shot examples. The *Code Clustering* pattern can be combined with the *Few-shot Code Example Generator* pattern in Section 4.5 to create code samples that demonstrate the desired property-based clustering and then use them in the *Code Clustering* pattern for in-context learning of the property.

**Consequences** The *Code Clustering* pattern can dramatically improve the quality of LLM-produced code. Unless the LLM is told otherwise, its code will solve the problem at hand and often does not solve structuring problems, such as

separating pure and impure functions, that it has not been asked to solve. The *Code Clustering* pattern surfaces a key issue in LLM software engineering, *i.e.*, an LLM’s output is only as good as the prompt it is given. Implicit knowledge (*e.g.*, the project requires code that exhibits certain clustering properties) will not be known to the LLM unless this information is provided to it in a prompt.

The *Code Clustering* pattern depends crucially on understanding implicit requirements, such as coding conventions, architectural constraints, separation of concerns, etc. An LLM may be unaware of these implicit requirements unless users explicitly mention them in the prompt. Making the LLM understand how the implicit requirements may require substantial space in the prompt. LLMs have limitations in learning and understanding the semantics behind code structuring and clustering that are not yet fully understood, so they may not always be capable of autonomously defining meaningful clusters even when given significant context.

## 5.2 The Intermediate Abstraction Pattern

**Intent and Context** The *Intermediate Abstraction* pattern instructs an LLM to not tightly couple some aspects of generated code by introducing an intermediate abstraction. This intermediate abstraction helps ensure that certain aspects of the code can be changed more easily at a later point in time. Abstraction and modularity are fundamental components of high-quality maintainable and reusable code. Code should be written in a way that isolates cohesive concepts into individual classes or methods so that edits can be isolated in scope. In addition, when working with an LLM, refactoring existing code is easier if refactorings can be isolated to a single function or method that needs to be modified, replaced, or added.

**Motivation** Be default, LLMs often generate code that is highly procedural and directly translates the requirements spelled out in the prompt into code. In particular, the implementation may not have sufficient abstraction or modularity, making it hard to maintain. Moreover, as an LLM is continually prompted to add features to the code, it may produce increasingly long methods with little separation of concepts into packages, classes, or other abstractions that facilitate long-term maintainability.

**Structure and Key Ideas** The fundamental contextual statements are as follows:

### Intermediate Abstraction Pattern

1. If you write or refactor code with property X
2. that uses other code with property Y
3. (Optionally) Define property X
4. (Optionally) Define property Y
5. Insert an intermediate abstraction Z between X and Y
6. (Optionally) Abstraction Z should have these properties

**Example Implementation** A sample implementation of the *Code Clustering* pattern is shown below:

”Whenever I ask you to write code, I want you to separate the business logic as much as possible from any underlying third-party libraries. Whenever business logic uses a third-party library, please write an intermediate abstraction that the business logic uses instead so the third-party library could be replaced with an alternate library if needed.”

A common risk in software is third-party libraries since their dependencies are not directly under the control of a developer, thereby creating project risk. For example, a third-party library developer introduce changes to its dependencies that make it hard to incorporate into an existing project and limit access to future security updates in the new version. This example implementation uses the LLM to insert an intermediate abstraction to mitigate this type of risk.

**Consequences** The *Code Clustering* pattern can aid in producing high-quality code. By explicitly instructing an LLM to structure its output in a certain way (*e.g.*, separating pure and impure functions), the resulting code can adhere to best practices and be easier to understand and maintain. Another benefit of this pattern is that it can help ensure consistency, particularly in large projects where multiple LLMs or even human developers might be involved. Consistent code is more predictable and easier to understand.

However, the *Code Clustering* pattern may not be able to design a good abstraction simply from analysis of a single third-party library that provides a given capability. For example, different dependencies may have different fundamental architectures and interfaces. One way to address this heterogeneity is to leverage the *Few-shot Example Generator* pattern in Section 4.5 to create examples of other comparable third-party libraries and their usage and then ask the LLM to refactor the interface so it can be implemented with any alternatives.

### 5.3 The Principled Code Pattern

**Intent and Context** The *Principled Code* pattern uses well-known names for coding principles to describe the desired code structure without having to

explicitly describe each individual design rule. For example, an organization may want to ensure that their code follows SOLID [15] design principles.<sup>2</sup> The goal is to ensure that generated, refactored, and reviewed code adheres to the expected design principles.

**Motivation** Writing software with good design characteristics is important to maintain a code base effective. However, developers may not be able to easily articulate and specify all the rules and patterns for what constitutes good design. Many articles and books have been written to explain how good design practices can be applied to different languages, frameworks, platforms, etc. The motivation for the *Principled Code* pattern is to enable developers to define these rules in natural language, as long as they know the commonly accepted name of the design method.

**Structure and Key Ideas** The fundamental contextual statements are as follows:

**Principled Code Pattern**

1. Within this scope
2. Generate, refactor, or create code to adhere to named Principle X

**Example Implementation** A sample implementation of the *Principled Code* pattern is shown below:

”From now on, whenever you write, refactor, or review code, make sure it adheres to SOLID design principles.”

This example uses the SOLID design principles as the desired design guidelines. This named design methodology informs the LLM of the underlying principles that code it generates should follow.

**Consequences** The *Principled Code* pattern works best with LLMs trained on a substantial volume of written material that explains the application of the named principle to a range of code bases. The more well-known the design principle(s), the more examples the LLM will likely have been trained on. The availability of training examples is particularly important for less mainstream languages or languages with more uncommon designs, such as Prolog or Clojure.

This situation is similar to the *Persona Pattern* presented in our prior work [24], where users describe the desired output using a well-known name. A consequence

---

<sup>2</sup> The SOLID code refers to software that applies the following design principles: (1) Single responsibility, (2) Open-closed, (3) Liskov substitution, (4) Interface segregation, and (5) Dependency inversion.

of the *Principled Code* pattern is that it only works with well-known named descriptions of code qualities that existed before the LLM was trained. Newer coding or design styles that came after the training date will not be accessible through this pattern. However, other approaches could be used to leverage in-context learning and few-shot examples to incorporate these inaccessible named coding or design styles.

#### 5.4 The Hidden Assumptions Pattern

**Intent and Context** The *Hidden Assumptions* pattern has an LLM identify and describe any assumptions made in a section of code. The pattern helps users identify these assumptions or remind them of assumptions they may have forgotten about. By showing key assumptions from the code to users, the LLM can help ensure users account for these assumptions in their decisions related to their code.

**Motivation** Any code, regardless if it is produced by a human or LLM, may have hidden assumptions that user must understand. If users are unaware of these assumptions, however, they may use, modify, or otherwise leverage the code incorrectly. Hidden assumptions are particular risky for LLM-generated code, where users may have less familiarity with what is being produced for them.

**Structure and Key Ideas** The fundamental contextual statements are as follows:

Hidden Assumptions Pattern
1. Within this scope
2. List the assumptions that this code makes
3. (Optionally) Estimate how hard it would be to change these assumptions or their likelihood of changing

**Example Implementation** Several sample implementations of the *Hidden Assumptions* pattern are shown below, starting with this one:

”List the assumptions that this code makes and how hard it would be to change each of them given the current code structure.”

This first example focuses on listing assumptions that may be hard to change in the future. This refinement of the pattern helps make developers aware of liabilities in the code with respect to future change. If one of the assumptions is hard to change—but developers expect this aspect will need to change—they can request the LLM to refactor it to remove this flawed assumption.

A second example of the *Hidden Assumptions* pattern shows how it can be used to aid in refactoring code from being tightly-coupled to an underlying database:

”List the assumptions in this code that make it hard to change from a MongoDB database to MySQL.”

With this example, the LLM will list assumptions that are tightly coupling to a specific database. User could then take this list and use it as the basis for refactoring, *e.g.*, by asking the LLM to refactor the code to eliminate the listed assumptions.

**Consequences** The *Hidden Assumptions* pattern enables the discovery of pre-suppositions or hidden assumptions present in the code. These insights can be invaluable in understanding the background context and underlying functioning of the code in question. Uncovering hidden assumptions of the code helps ensure that it is used, modified, or referenced correctly, mitigating the risk of misuse due to misunderstood or overlooked assumptions.

However, the *Hidden Assumptions* pattern may not identify all hidden assumptions in the code. For example, there may be code outside of what is in the context provided to the LLM that is needed to identify the assumption. The risk of this pattern is that developers will take it as a source of truth for all assumptions in the code, rather than flagging of some possible assumptions for consideration by developers.

## 5.5 The Pseudo-code Refactoring Pattern

**Intent and Context** The *Pseudo-code Refactoring* pattern give users fine-grained control over the algorithm, flow, or other aspects of the code, while not requiring explicit specification of details. It also allows users to define pseudo-code for one or more details of generated or refactored code. The LLM is expected to adapt the output to fit the pseudo-code template while ensuring the code is correct and runnable.

**Motivation** In many cases, users may have strong opinions or specific goals in the refactoring or generation of code that are tedious to describe—and duplicative of an LLM’s work—if were necessary to type the exact code structures they wanted. In particular, the benefit of using an LLM is reduced if developers must do as much coding work as the LLM to specify what they want. The motivation of the *Pseudo-code Refactoring* pattern is to provide a middle ground that allows greater control over code aspects without explicit coding and considering all the details.

**Structure and Key Ideas** The fundamental contextual statements are as follows:

**Pseudo-code Refactoring Pattern**

1. Refactor the code
2. So that it matches this pseudo-code
3. Match the structure of the pseudo-code as closely as possible

**Example Implementation** A sample implementation of the *Pseudo-code Refactoring* pattern is shown below:

”Refactor the following code to match the following psuedo-code. Match the structure of the pseudo-code as closely as possible.

```
files = scan_features()
for file in files:
    print file name
for file in files:
    load feature
    mount router
create_openapi()
main():
    launch app"
```

In this example, the prompt is asking the LLM to refactor a much larger body of code to match the structure of the pseudo-code. The pseudo-code defines the outline of the code, but not the details of how individual tasks are accomplished. In addition, the pseudo-code does not provide exact traceability to which lines are part of the described functionality. The LLM determines what the intent of the refactoring is and how to map it into the current code provided to it.

**Consequences** The *Pseudo-code Refactoring* pattern enables more precise control over code generation or refactoring. It empowers users to influence the algorithm’s flow, structure or other aspects without going into exhaustive details. The pattern also boosts efficiency by enabling users to outline pseudo-code structures that the LLM can populate. This pseudo-code reduces duplicate work and saves time since developers need not program the full code structure that they desire.

The *Pseudo-code Refactoring* pattern can trigger more substantial refactoring than what is outlined in the pseudo-code. For example, rewriting the code to match the pseudo-code may require the LLM to remove a method and split its code between two other methods. However, removing the method could then change the public interface of the code.

## 5.6 The Data-guided Refactoring Pattern

**Intent and Context** The *Data-guided Refactoring* pattern allow users to refactor existing code to use data with a new format. Rather than specifying the exact logic changes needed to use the new format, users can provide the new format schema to the LLM and ask the LLM to make the necessary changes. This pattern helps automate code refactoring for the common task of incorporating changes to data formats.

**Motivation** Refactoring code to use a new input or output data structure can be tedious. When communicating with an LLM, explaining the explicit refactoring steps to may also require more time than actually conducting the needed steps. The *Data-guided Refactoring* pattern provides a concise way of explaining to the LLM what refactoring is needed. Since changing data formats can have such a large-scale impact on a system, this pattern can automate these refactorings, potentially reducing costs and accelerating overall system development.

**Structure and Key Ideas** The fundamental contextual statements are as follows:

### Data-guided Refactoring Pattern

1. Refactor the code
2. So that its input, output, or stored data format is X
3. Provide one or more examples of X

**Example Implementation** An example implementation of the *Data-guided Refactoring* pattern is shown below:

```
"Let's refactor execute_graph() so that graph has the following format
{'graph':{ ...current graph format... }, 'sorted_nodes': { 'a': ['b','c'...],...}}"
```

This example asks the LLM to refactor a method to use a different format for the graph. In the example, the specific use of the graph format is not defined, but could potentially be input, output, or internal to the method. All the different uses of the graph would be supportable by the pattern. In addition, the implementation uses ellipses to indicate portions of the data structure, which allows the LLM to fill in user intent with concrete details.

**Consequences** The *Data-guided Refactoring* pattern reduces the manual effort to specify refactoring for many types of code changes necessitated by a change in data format. In many cases, the refactoring can be completely automated through this process, or at least bootstrapped, thereby accelerating and potentially reducing the cost of changing data formats.

While LLMs can operate with many different schemas and refactor code accordingly, they may struggle with complex or niche data formats different from what the LLM saw in its training data. Developers may therefore need to provide additional guidance or instruction in these cases. Moreover, an LLM might not understand the full implications of the data format change and may not refactor all necessary parts of the code, leading to an inconsistent codebase that uses multiple data formats at different points in the code.

## 6 Related Work

Software patterns [10, 20] have been studied extensively and shown their value in software engineering. Software design patterns have also been specialized for other types of non-traditional uses, such as designing smart contracts [26, 28]. Prompt design patterns for software engineering are complementary to these, although not focused on the design of the system itself, but on the interactions with an LLM to produce and maintain software-reliant systems over time.

Prompt engineering is an active area of study and the importance of prompts is well understood [7]. Many problems cannot be solved by LLMs unless prompts are structured correctly [9]. Some work has specifically looked at prompting approaches to help LLMs learn to leverage outside tooling [27]. Our work complements these approaches, focusing on specific patterns of interaction that can be used to tap into LLM capabilities to solve specific problems in software engineering.

Much discussion on LLM usage in software engineering to date has centered on the use of LLMs for code generation and the security and code quality risks associated with that usage. For example, Asare et al. [2] compared LLM code generation to humans from a security perspective. Other research has examined the quality of generated answers and code from LLMs [5, 9, 11, 16] and interaction patterns for fixing bugs [21, 25]. Our research draws inspiration from these explorations and documents specific patterns that can be used to improve code quality and help reduce errors. Moreover, as more prompt patterns are developed, different patterns can be quantitatively compared to each other for effectiveness in solving code quality issues.

## 7 Concluding Remarks

Ever since ChatGPT was released to the public in November 2022, much attention [1, 2, 18] has focused on the mistakes that LLMs make when performing software engineering tasks particularly with respect to generating code with defects and/or security vulnerabilities. As shown in this chapter, however, prompt patterns can be used to help alleviate many of these mistakes and reduce errors. Moreover, prompt patterns can tap into LLM capabilities that are hard to automate using existing technologies, including simulating a system based on requirements, generating an API specification, and pointing out assumptions in code.,

The following are lessons learned thus far from our work on applying ChatGPT to automate common software engineering tasks:

- **Prompt engineering is crucial for unlocking the full capabilities of LLMs for software engineering tasks.** The prompt patterns described in this chapter codify effective prompt design techniques that can help address common software engineering challenges. Thoughtful prompt design is key to tapping into ChatGPT’s strengths.
- **Prompt patterns enable rapid exploration and experimentation throughout the software life-cycle.** Patterns like the *API Generator*, *API Simulator*, and *Architectural Possibilities* allow developers to experiment rapidly with multiple designs and approaches early in the life-cycle. This agility can accelerate development and lead to better solutions.
- **Integrating prompt patterns together into pattern catalogs can further enhance their effectiveness.** This chapter discusses chaining prompt patterns together, such as combining the *Requirements Simulator* and *Visualization Generator* patterns. These sequences and combinations of patterns can build upon each other to accomplish more complex goals.
- **The depth of capabilities of LLMs, such as ChatGPT, is not widely or fully understood or appreciated.** LLMs hold immense potential for helping to automate common tasks throughout the software engineering life-cycle. Many LLM capabilities have the potential to accelerate software engineering, not just by generating code, but by making rapid experimentation at many different levels of abstraction possible. A key to leveraging these capabilities is to codify an effective catalog of prompts and guidance on how to combine them at different stages of the software life-cycle to improve software engineering.
- **Significant human involvement and expertise is currently necessary to leverage LLMs effectively for automating common software engineering tasks.** The tendency of ChatGPT to “hallucinate” confidently and enthusiastically when generating incorrect output requires close scrutiny from human users at this point. While prompt patterns can help mitigate some of these issues, much further work is needed on other aspects of prompt engineering (such as quality assurance and versioning) to ensure output of LLMs is accurate and helpful in practice.

We encourage readers to test the prompt patterns described in this chapter by using ChatGPT to replicate our findings in their own domains and environments.

## References

1. Github copilot · your ai pair programmer, [github.com/features/copilot](https://github.com/features/copilot)
2. Asare, O., Nagappan, M., Asokan, N.: Is github’s copilot as bad as humans at introducing vulnerabilities in code? arXiv preprint arXiv:2204.04741 (2022)
3. Bang, Y., Cahyawijaya, S., Lee, N., Dai, W., Su, D., Wilie, B., Lovenia, H., Ji, Z., Yu, T., Chung, W., et al.: A multitask, multilingual, multimodal evaluation of chatgpt on reasoning, hallucination, and interactivity. arXiv preprint arXiv:2302.04023 (2023)

4. Bommasani, R., Hudson, D.A., Adeli, E., Altman, R., Arora, S., von Arx, S., Bernstein, M.S., Bohg, J., Bosselut, A., Brunskill, E., et al.: On the opportunities and risks of foundation models. arXiv preprint arXiv:2108.07258 (2021)
5. Borji, A.: A categorical archive of chatgpt failures. arXiv preprint arXiv:2302.03494 (2023)
6. Carleton, A., Klein, M.H., Robert, J.E., Harper, E., Cunningham, R.K., de Niz, D., Foreman, J.T., Goodenough, J.B., Herbsleb, J.D., Ozkaya, I., Schmidt, D.C.: Architecting the future of software engineering. Computer **55**(9), 89–93 (2022)
7. van Dis, E.A., Bollen, J., Zuidema, W., van Rooij, R., Bockting, C.L.: Chatgpt: five priorities for research. Nature **614**(7947), 224–226 (2023)
8. Feiler, P., Sullivan, K., Wallnau, K., Gabriel, R., Goodenough, J., Linger, R., Longstaff, T., Kazman, R., Klein, M., Northrop, L., Schmidt, D.: Ultra-Large-Scale Systems: The Software Challenge of the Future. Software Engineering Institute, Carnegie Mellon University (2006)
9. Frieder, S., Pinchetti, L., Griffiths, R.R., Salvatori, T., Lukasiewicz, T., Petersen, P.C., Chevalier, A., Berner, J.: Mathematical capabilities of chatgpt. arXiv preprint arXiv:2301.13867 (2023)
10. Gamma, E., Johnson, R., Helm, R., Johnson, R.E., Vlissides, J.: Design patterns: elements of reusable object-oriented software. Pearson Deutschland GmbH (1995)
11. Jalil, S., Rafi, S., LaToza, T.D., Moran, K., Lam, W.: Chatgpt and software testing education: Promises & perils. arXiv preprint arXiv:2302.03287 (2023)
12. Kendar, R.: Httpansweringmachine. github.com/kendarorg/HttpAnsweringMachine (2022), accessed on March 11, 2023
13. Krochmalski, J.: IntelliJ IDEA Essentials. Packt Publishing Ltd (2014)
14. Liu, P., Yuan, W., Fu, J., Jiang, Z., Hayashi, H., Neubig, G.: Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing. ACM Computing Surveys **55**(9), 1–35 (2023)
15. Marshall, D., Bruno, J.: Solid code. Microsoft Press (2009)
16. Nair, M., Sadhukhan, R., Mukhopadhyay, D.: Generating secure hardware using chatgpt resistant to cwe. Cryptology ePrint Archive (2023)
17. OpenAPI Initiative: Openapi specification. <https://www.openapis.org/> (2021), accessed on March 11, 2023
18. Pearce, H., Ahmad, B., Tan, B., Dolan-Gavitt, B., Karri, R.: Asleep at the keyboard? assessing the security of github copilot's code contributions. In: 2022 IEEE Symposium on Security and Privacy (SP). pp. 754–768. IEEE (2022)
19. Schmidt, D.C.: Guest Editor's Introduction: Model-Driven Engineering. IEEE Computer **39**(2), 25–31 (2006)
20. Schmidt, D.C., Stal, M., Rohnert, H., Buschmann, F.: Pattern-oriented software architecture, patterns for concurrent and networked objects. John Wiley & Sons (2013)
21. Sobania, D., Briesch, M., Hanna, C., Petke, J.: An analysis of the automatic bug fixing performance of chatgpt. arXiv preprint arXiv:2301.08653 (2023)
22. Wadler, P.: The essence of functional programming. In: Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. pp. 1–14 (1992)
23. Wang, Y., Yao, Q., Kwok, J.T., Ni, L.M.: Generalizing from a few examples: A survey on few-shot learning. ACM computing surveys (csur) **53**(3), 1–34 (2020)
24. White, J., Fu, Q., Hays, S., Sandborn, M., Olea, C., Gilbert, H., Elnashar, A., Spencer-Smith, J., Schmidt, D.C.: A prompt pattern catalog to enhance prompt engineering with chatgpt. arXiv preprint arXiv:2302.11382 (2023)

25. Xia, C.S., Zhang, L.: Conversational automated program repair. arXiv preprint arXiv:2301.13246 (2023)
26. Xu, X., Pautasso, C., Zhu, L., Lu, Q., Weber, I.: A pattern collection for blockchain-based applications. In: Proceedings of the 23rd European Conference on Pattern Languages of Programs. pp. 1–20 (2018)
27. Yao, S., Zhao, J., Yu, D., Du, N., Shafran, I., Narasimhan, K., Cao, Y.: React: Synthesizing reasoning and acting in language models. arXiv preprint arXiv:2210.03629 (2022)
28. Zhang, P., White, J., Schmidt, D.C., Lenz, G.: Applying software patterns to address interoperability in blockchain-based healthcare apps. CoRR **abs/1706.03700** (2017), <http://arxiv.org/abs/1706.03700>