Promptware Engineering: Software Engineering for LLM Prompt Development

ZHENPENG CHEN, Nanyang Technological University, Singapore
CHONG WANG, Nanyang Technological University, Singapore
WEISONG SUN, Nanyang Technological University, Singapore
GUANG YANG, Microsoft, China
XUANZHE LIU, Peking University, China
JIE M. ZHANG, King's College London, United Kingdom
YANG LIU, Nanyang Technological University, Singapore

Large Language Models (LLMs) are increasingly integrated into software applications, with prompts serving as the primary 'programming' interface to guide their behavior. As a result, a new software paradigm, *promptware*, has emerged, using natural language prompts to interact with LLMs and enabling complex tasks without traditional coding. Unlike traditional software, which relies on formal programming languages and deterministic runtime environments, promptware is based on ambiguous, unstructured, and context-dependent natural language and operates on LLMs as runtime environments, which are probabilistic and non-deterministic. These fundamental differences introduce unique challenges in prompt development. In practice, prompt development is largely ad hoc and experimental, relying on a time-consuming trial-and-error process — a challenge we term the '*promptware crisis*.' To address this, we propose *promptware engineering*, a new methodology that adapts established software engineering principles to the process of prompt development. Building on decades of success in traditional software engineering, we envision a systematic framework that includes prompt requirements engineering, design, implementation, testing, debugging, and evolution. Unlike traditional software engineering, our framework is specifically tailored to the unique characteristics of prompt development. This paper outlines a comprehensive roadmap for promptware engineering, identifying key research directions and offering actionable insights to advance LLM-based software development.

Additional Key Words and Phrases: Promptware Engineering, Software Engineering, Prompt, Large Language Model

arXiv:submit/6250816 [cs.SE] 4 Mar 2025

1 Introduction

Large Language Models (LLMs), such as GPT [7], LLaMA [40], and DeepSeek [10], are increasingly integrated into software applications across diverse domains [15, 29, 44]. Major technology companies, including Microsoft, Google, Amazon, and Apple, have all integrated LLMs into their software products [34], reaching millions of users.

In LLM-based software, prompts serve as the primary 'programming' interface, directly shaping the behavior and outputs of LLMs [39]. Recognizing their central role, researchers have described LLM-based software as prompt-powered software [30], while practitioners have introduced template-based approaches, such as the Liquid prompt template [4] and LangChain prompt template [3], to support prompt programming. This shift has led to the emergence of *promptware* [23], a new paradigm where natural language prompts replace traditional code, enabling complex tasks to be executed through direct interaction with LLMs. As LLMs continue to advance, promptware is likely to become the dominant paradigm in software development, with prompts replacing traditional coding as the primary method for creating and managing LLM-based software.

Authors' Contact Information: Zhenpeng Chen, zhenpeng.chen@ntu.edu.sg, Nanyang Technological University, Singapore; Chong Wang, chong.wang@ntu.edu.sg, Nanyang Technological University, Singapore; Weisong Sun, weisong.sun@ntu.edu.sg, Nanyang Technological University, Singapore; Guang Yang, guangyang@microsoft.com, Microsoft, China; Xuanzhe Liu, liuxuanzhe@pku.edu.cn, Peking University, China; Jie M. Zhang, jie.zhang@kcl.ac.uk, King's College London, United Kingdom; Yang Liu, yangliu@ntu.edu.sg, Nanyang Technological University, Singapore.

Promptware fundamentally differs from traditional software in two key aspects: language and runtime environment. Unlike structured programming languages with strict syntax and deterministic behavior, prompts are written in natural language, which is flexible, context-dependent, and ambiguous. This makes formalizing prompt construction and analysis challenging. Additionally, while traditional software typically relies on deterministic runtime environments, promptware uses probabilistic, non-deterministic LLMs as runtime environments, introducing unique challenges such as human-like behaviors, unclear capability boundaries, undefined error handling, and uncertain execution control. These complexities make prompt development particularly difficult.

Prompt engineering has been a widely adopted solution for prompt development [38]. OpenAI defines it as 'designing and optimizing input prompts to effectively guide a language model's responses' [5]; Meta calls it 'a technique used in natural language processing to improve the performance of the language model by providing more context and information about the task at hand' [6]. These organizations have published official guidelines to support prompt engineering [1, 6], but they share a critical limitation: they are output-centric and lack systematic methodologies for prompt development.

In practice, prompt engineering often relies on ad hoc, experimental approaches [17, 24, 30, 33, 34, 36, 39], especially as LLM-based software becomes increasingly complex—what we refer to as the '*promptware crisis*.' Existing studies [17, 30, 33, 36] show that prompt engineering is largely trial-and-error, time-consuming, and challenging, with even experienced engineers facing difficulties. This highlights the urgent need for a systematic, development-centric framework to move beyond the ad hoc approaches and address the growing challenges of the promptware crisis.

In this paper, we introduce *promptware engineering*, a new methodology that applies software engineering (SE) principles to prompt development. This vision is driven by two key rationales: (1) As LLMs become integral to an expanding range of software applications, prompts have emerged as crucial software components. Thus, promptware has surfaced as an evolving software paradigm, with prompt development increasingly recognized as a new form of programming [30]. This shift positions prompt development as a vital aspect of SE. (2) Current prompt development practices are often ad hoc and experimental, driven by unique complexities. However, decades of SE advancements highlight the value of systematic approaches to manage complexity, ensure quality, and support iterative improvements, underscoring the potential to transform prompt development from an experimental practice into a structured and disciplined process.

Realizing the vision of promptware engineering requires integrating core SE activities, specifically tailored to the unique demands of prompt development. These activities include prompt requirements engineering, design, implementation, testing, debugging, and evolution. To support them, we emphasize the need for innovative research, tools, and automation throughout the prompt development lifecycle. This paper discusses key challenges and highlights promising research opportunities in promptware engineering.

2 Preliminaries

This section defines key terminologies and situates our vision within the context of existing research.

2.1 Terminologies

LLM. LLMs are advanced computational models designed to understand and generate human language, distinguished by their massive parameter sizes and ability to learn from large and diverse datasets [12]. Common examples include GPT [7], LLaMA [40], Claude [2], and DeepSeek [10].

LLM-based software. LLM-based software refers to software applications that integrate LLMs as components to perform diverse tasks [33]. It typically includes one or more LLMs, prompts for interacting with them, and additional supporting code and components.

Prompt. Prompts are natural language instructions or queries given to LLMs that define the context, task, or expected behavior [39]. They allow LLMs to perform downstream tasks without modifying their parameters and are essential components of LLM-based software.
Promptware. Promptware is a software paradigm that uses natural language prompts to interact with LLMs [23], enabling complex tasks without traditional coding.
Promptware engineering. Promptware engineering is the methodology of applying SE principles to the development of prompts.

It can be viewed as a new methodology that bridges prompt engineering and SE. It extends prompt engineering by specifying the engineering principles that guide prompt development, offering a systematic framework for creating effective and reliable interactions with LLMs. Simultaneously, it broadens the scope of SE by introducing prompts as a new software component in the context of emerging LLM-based software.

2.2 Related Work

There has been numerous research combining prompt engineering and SE. Most of these works focus on prompt engineering for SE. The widespread adoption of LLMs in SE tasks has sparked significant interest in prompt engineering techniques aimed at improving the effectiveness of LLMs in these scenarios. Wang et al. [43] conducted a comprehensive survey on the use of LLMs in software testing, a critical SE activity. Their findings revealed that about two-thirds of studies in this domain employ prompt engineering. Fan et al. [18] and Hou et al. [25] extended the scope of the surveys to encompass the entire SE lifecycle. Fan et al. [18] reviewed common prompt engineering strategies applied to tasks such as code generation, software testing, and maintenance. Hou et al. [25] identified eight prompt engineering techniques currently employed in the LLM for SE domain. Alshahwan et al. [8] proposed a vision for assured LLM-based SE, outlining how search-based SE approaches can be leveraged to optimize prompts in SE activities. Recently, the rise of LLM-based agents in SE has further highlighted the importance of prompt engineering. These techniques have been widely adopted in constructing agents to enhance their performance across various SE tasks [31].

In contrast, SE for prompt engineering has not been well explored. Existing studies primarily focus on empirical investigations into the challenges software engineers face during prompt engineering. Mailach et al. [33] and Parnin et al. [36] studied challenges faced by software practitioners in developing LLM-based applications, and identified prompt engineering as a key concern. Practitioners highlighted issues such as the experimental nature of prompting, as well as challenges related to context length, computational costs, and unpredictable changes in prompts [33]. They also noted that designing and managing prompts efficiently is time-consuming and resource-constrained [36]. Similarly, Dolata et al. [17] found freelancers struggle with trial-and-error prompting cycles. They reported that subtle differences in prompts resulted in inconsistent outputs and identified challenges in managing experimentation costs. Liang et al. [30] conceptualized prompts as a new type of program and conducted interviews to understand how developers integrate them into software. Their findings revealed that prompt programming is a rapid, unsystematic process, significantly different from traditional software development. Tafreshipour et al. [39] studied prompt evolution in real-world projects and reported that only 21.9% of prompt changes are documented. These changes often result in logical inconsistencies and misalignments between prompts and LLM responses. Nahar et al. [34] interviewed product teams at Microsoft and confirmed numerous challenges in LLM-based software development, including those specifically related to prompt engineering. These studies emphasize the pressing need to transition from the current experimental approach to a systematic framework for prompt development. Recently, Hassan et al. [24] identified crafting effective prompts as one of the ten key SE challenges and proposed prompt IDEs as a potential

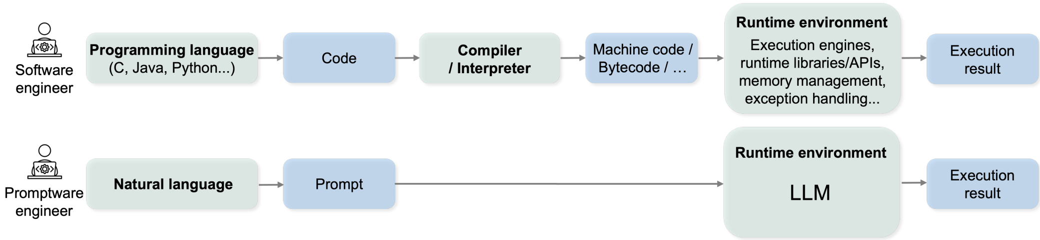

Fig. 1. Comparison of the traditional software paradigm and promptware.

Table 1. Traditional programming language vs. natural language.

Aspect	Programming Language	Natural Language
C1. Structure and Formalism	Highly structured with rigorous syntax and well-defined semantics.	Unstructured, flexible, and open-ended.
C2. Explicitness and Determinism	Explicit; deterministic behavior.	Ambiguous and context-dependent; probabilistic behavior.
C3. Correctness and Quality	Explicit correctness defined by formal specifications; syntax errors are fatal.	No universal correctness; typos or grammatical errors may not cause failures but can subtly alter meaning.

solution. In contrast, this paper presents a vision for promptware engineering, encompassing the entire SE lifecycle for prompt development.

3 Traditional Software Paradigm vs. Promptware

Before exploring the research opportunities in promptware engineering, we first compare promptware with the traditional software paradigm to highlight its unique characteristics.

Figure 1 illustrates the comparison. Traditionally, software engineers write code using programming languages such as C, Java, and Python. Depending on the language, a compiler or interpreter translates the code into machine code, bytecode, or an intermediate representation. The translated code is then executed within a runtime environment, which typically includes components such as execution engines, runtime libraries, memory management systems, and exception handling mechanisms.

In contrast, promptware engineers write prompts primarily using natural language, eliminating the need for traditional compilation or interpretation. Instead, an LLM serves as the runtime environment, interpreting the input prompt and generating responses based on probabilistic reasoning and pre-trained knowledge, often in real-time.

This comparison indicates that the unique characteristics of promptware primarily arise from the distinct nature of its language and runtime environment. In the following, we outline these specific characteristics, labeled as C1, C2, ..., C10. A summary of the comparison between promptware and the traditional software paradigm in terms of language is presented in Table 1, while the comparison regarding the runtime environment is summarized in Table 2.

3.1 Traditional Programming Language vs. Natural Language

C1. Structure and Formalism: Traditional programming languages are highly structured, with rigorous syntax and well-defined semantics that enforce precise coding rules. In contrast, natural language is inherently unstructured, flexible, and open-ended, making it difficult to establish formalized rules for prompt construction.

Table 2. Traditional runtime environment vs. LLM-as-runtime environment.

Aspect	Traditional Runtime Environment	LLM-as-Runtime Environment
C4. Determinism	Same input typically yields the same output.	Same prompt can yield different outputs.
C5. Human-Like Characteristics	Executes code mechanically following predefined logic.	Exhibits human-like characteristics (e.g., contextual reasoning and social intelligence).
C6. Capability Boundaries	Well-defined capability boundaries; engineers understand execution rules.	Unclear, evolving capability boundaries; execution functions as a black box.
C7. Error Handling	Deterministic error handling, with clear error messages, stack traces, debugging mechanisms.	Implicit and unpredictable fault tolerance, without explicit error signals.
C8. Execution Control	Precise execution control: step through code, inspect variables, use debuggers.	Indirect control: adjust prompts heuristically experimentally.
C9. Access Control	Strict access control: permissions, memory protections, sandboxing.	Limited access control: vulnerable to security risks, lack of rigid execution boundaries.
C10. Memory Management	Explicit memory management: direct control over allocation and deallocation.	No persistent memory: contextual processing without long-term memory unless external memory mechanisms are used.

C2. Explicitness and Determinism: Traditional programming languages are explicit and deterministic, meaning the same code typically produces the same output. This determinism enables well-established techniques for analyzing aspects such as data flow, control flow, and dependencies. In contrast, natural language is ambiguous and context-dependent, relying on prior knowledge, pragmatics, and contextual cues rather than rigid rules. Its probabilistic nature makes it difficult to ensure consistent and reliable prompt outcomes, as well as to systematically analyze how prompts influence model outputs.
C3. Correctness and Quality: In traditional programming, correctness is explicitly defined through formal specifications, and both functional and non-functional quality attributes can be systematically measured. Even minor syntax errors can lead to failures or bugs. In contrast, natural language lacks a universal standard for correctness but follows conventions and guidelines (such as grammar rules and clarity of intent) that affect a prompt's effectiveness. While minor typos or grammatical inconsistencies may not always cause immediate failures, they can subtly alter meaning, potentially leading to unpredictable or undesired outcomes.

3.2 Traditional Runtime Environment vs. LLM-as-Runtime Environment

C4. Determinism: Traditional runtime environments are deterministic, meaning the same input typically produces the same output under identical conditions. Execution follows well-defined rules dictated by the programming language, system architecture, and compiler or interpreter behavior. In contrast, LLMs generate responses probabilistically based on learned statistical patterns, resulting in inherent non-determinism. The same prompt can produce different outputs across executions, posing challenges for reproducibility, reliability, and consistency in promptware engineering.
C5. Human-Like Characteristics: Traditional runtime environments execute code mechanically without subjective interpretation. In contrast, LLMs exhibit human-like characteristics, such as contextual reasoning, emotional alignment, and social intelligence. While these traits enhance flexibility, they introduce challenges in control, intent alignment, and reliability. LLMs may generate responses influenced by biases, emotional tone, or implicit assumptions, making promptware harder to stabilize, predict, and interpret.
C6. Capability Boundaries: Traditional runtime environments have well-defined capability boundaries, with execution governed by explicit rules and formal models. Engineers can analyze and predict system behavior based on well-documented specifications. In contrast, LLMs have

unclear and evolving capability boundaries, often exhibiting emergent behaviors. Moreover, LLM execution functions as a black box—developers lack direct visibility into how prompts are internally processed, making it difficult to predict, control, or explain outcomes.
C7. Error Handling: Traditional runtime environments handle errors deterministically, providing well-defined error messages and stack traces. Errors are explicit and follow strict handling protocols. In contrast, LLMs exhibit implicit and unpredictable fault tolerance. Instead of producing explicit error messages, they attempt to generate a response, which may be misleading, subtly incorrect, or entirely hallucinated. This absence of strict error signaling complicates promptware debugging and reliability assessment.
C8. Execution Control: Traditional runtime environments provide precise execution control, allowing engineers to step through code line by line, inspect variable states, and utilize debugging tools. In contrast, LLM execution is opaque—engineers can only influence behavior indirectly through prompt modifications. Debugging is heuristic and experimental, often relying on iterative prompt adjustments rather than structured debugging tools. This lack of systematic execution control makes diagnosing and fixing undesired behaviors more challenging.
C9. Access Control: Traditional runtime environments enforce strict access controls through permissions, memory protections, and sandboxing. In contrast, LLMs lack fine-grained access control mechanisms, making them vulnerable to security risks such as prompt injection and unintended leakage of sensitive information. Ensuring security in promptware is challenging due to LLMs' inability to enforce rigid execution boundaries or restrict unauthorized data access.
C10. Memory Management: In traditional runtime environments, memory management is explicit, with direct control over allocation, deallocation, and garbage collection, ensuring efficient resource handling. In contrast, LLMs lack persistent memory in the conventional sense and process inputs contextually, meaning they do not retain state across interactions unless external memory mechanisms (e.g., databases) are used. For promptware, this creates a challenge in maintaining continuity and coherence across prompts, especially for tasks requiring long-term dependency tracking or sustained context across multiple interactions.

4 Promptware Engineering: Roadmap

This section presents a roadmap for promptware engineering (as shown in Figure 2), covering key SE activities including requirements engineering, design, implementation, testing, debugging, and evolution. For each activity, we identify promising research opportunities. To facilitate reference, each opportunity is uniquely labeled with a counter, e.g., O1. Additionally, we specify which aspects of promptware's characteristics (C1 to C10) need to be considered in each opportunity.

4.1 Prompt Requirements Engineering

Requirements engineering (RE) serves as the foundation of software development, focusing on translating user needs into clear, actionable specifications. In the realm of promptware engineering, RE involves identifying the requirements and defining the specifications for prompts.
O1: LLM-driven prompt RE (C4, C5, C6, C9). In traditional RE, requirements are primarily derived from user needs. However, in promptware engineering, requirements must also account for LLM capabilities and constraints, which influence how prompts are designed, refined, and validated.

First, prompt requirements must align with an LLM's reasoning ability, language comprehension, and domain expertise. However, LLMs' capability boundaries (C6) are unpredictable and evolve, necessitating adaptive requirements that can be iteratively refined. Additionally, LLMs exhibit non-deterministic execution (C4), meaning the same prompt can produce different outputs, complicating the definition of consistent and testable requirements.

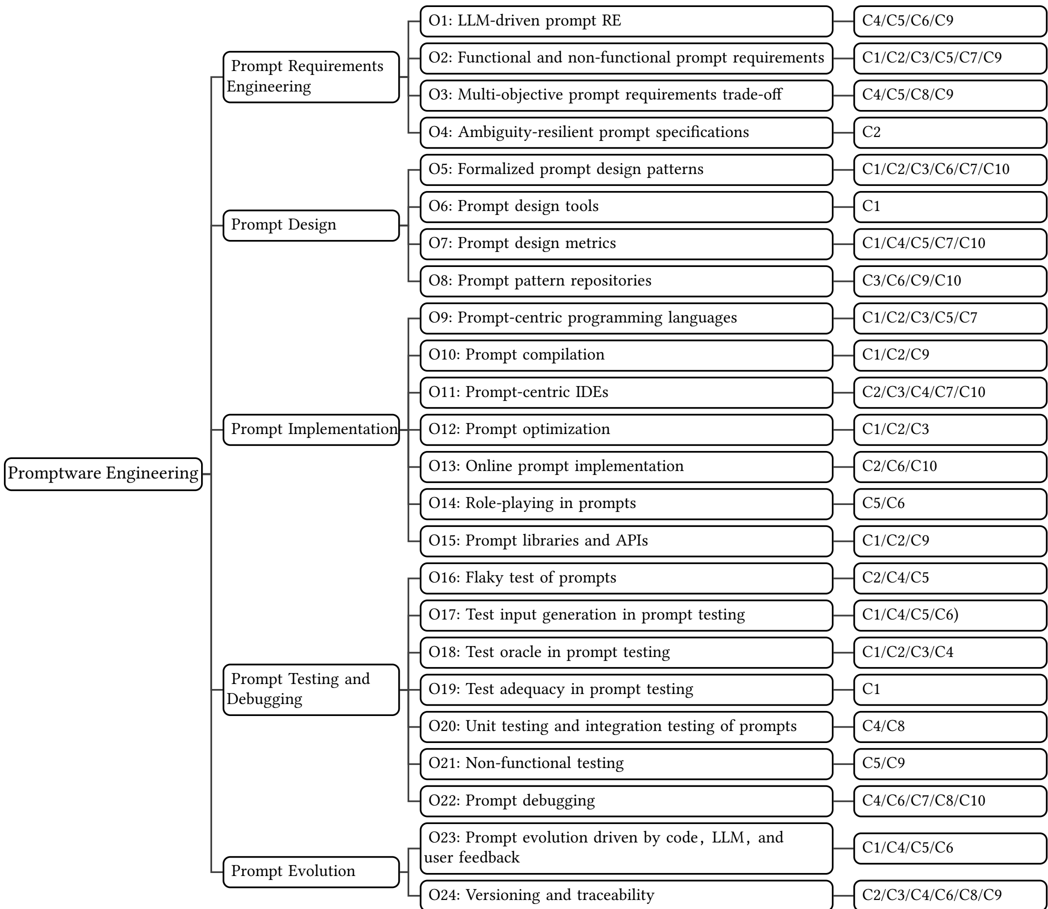

Fig. 2. Roadmap for promptware engineering, highlighting key activities alongside associated research opportunities (O1, O2, etc.) and the relevant promptware characteristics (C1 to C10) to be considered for each opportunity.

Second, LLMs exhibit human-like characteristics (C5), such as personality [21], biases [42], emotional alignment [26], and contextual reasoning [19]. These characteristics must be explicitly accounted for in prompt requirements, especially in culturally sensitive or ethical applications. For example, requirements should define tone, formality, and ethical safeguards to ensure appropriate and unbiased interactions.

Furthermore, prompt requirements must also consider security risks of LLMs (C9). Poorly designed prompts may inadvertently expose sensitive information or be vulnerable to adversarial manipulation.

To define effective prompt requirements, RE in promptware engineering can integrate insights from both computer and social sciences, fostering interdisciplinary collaboration among computer scientists, linguists, and psychologists.

O2: Functional and non-functional prompt requirements (C1, C2, C3, C5, C7, C9). In traditional SE, functional requirements define the explicit tasks a system must perform, while non-functional requirements describe quality attributes such as performance, security, and robustness.

This distinction remains essential in promptware engineering but requires adaptation due to the open-ended and ambiguous nature of natural language prompts (C1, C2).

Functional prompt requirements need to ensure clarity, specificity, and context-awareness. Prompts should be designed to explicitly communicate task objectives while minimizing ambiguities (C1, C2). Although deterministic correctness for prompts is infeasible, ensuring reliability remains a core objective (C3).

Non-functional requirements, in contrast, ensure that prompts produce secure, fair, efficient, and robust behaviors. Prompts should be resistant to bias amplification, adversarial manipulation, and ethical risks (C5, C9). Additionally, performance constraints, such as response latency and token efficiency, must be balanced against quality and interpretability. As LLMs do not follow traditional error-handling mechanisms (C7), prompt robustness must be evaluated under adversarial conditions to prevent failures that might otherwise go unnoticed.

Due to the probabilistic and evolving nature of LLMs, the definition, scope, and prioritization of functional and non-functional requirements must remain adaptable and be continuously refined through empirical validation.

O3: Multi-objective prompt requirements trade-off (C4, C5, C8, C9). Promptware engineering often requires balancing competing objectives, making trade-off analysis a critical aspect of RE. For instance, role-playing techniques, which guide LLMs to assume specific roles, can enhance response relevance but may also amplify pre-existing biases [28] (C5). Similarly, improving prompt clarity and specificity by providing detailed instructions enhances output quality but increases token consumption, impacting computational efficiency and cost.

Addressing these trade-offs necessitates a structured framework for systematically evaluating competing priorities. Since LLM execution is inherently non-deterministic (C4) and lacks direct execution control (C8), trade-off decisions need to rely on empirical testing and iterative refinement rather than formal verification. Additionally, prompt security (C9) must be integrated into trade-off considerations to prevent usability and performance optimizations from introducing vulnerabilities such as prompt injection attacks.

Developing systematic methodologies for balancing multiple prompt objectives remains a key research challenge, requiring the integration of multi-objective optimization techniques into promptware engineering workflows.

O4: Ambiguity-resilient prompt specifications (C2). Natural language prompts are inherently ambiguous (C2), yet precise specification is crucial for achieving predictable and reliable performance. Traditional software specification techniques are inadequate for promptware engineering because they rely on formal syntax and semantics, whereas prompts depend on natural language, which lacks the rigor necessary for deterministic validation. This challenge highlights the need for new specification approaches that mitigate ambiguity while preserving the adaptability required for effective prompt design.

One potential direction is the development of semi-formal or formal specification languages specifically tailored for prompts. These languages could integrate structured templates with natural language annotations to improve clarity and reduce ambiguity. For example, structured elements could define task objectives, constraints, and contextual dependencies, while natural language annotations provide flexibility and interpretability. A hybrid approach like this could enable automated validation and consistency checks, ensuring that prompts adhere to defined requirements and perform reliably across varying conditions and use cases.

4.2 Prompt Design

Software design defines the architecture, components, interfaces, and characteristics of a system to ensure it meets specified requirements. In the context of promptware engineering, design focuses on structuring and organizing prompts to achieve effective model interactions.

Current prompt design has primarily been driven by the AI community. Common prompt structures, such as zero-shot prompting (direct queries without examples) [45], few-shot prompting (providing in-context examples) [11], Chain-of-Thought (CoT) prompting (breaking down reasoning into intermediate steps) [46], and Retrieval-Augmented Generation (RAG) (incorporating external knowledge retrieval) [27], are widely used but developed empirically rather than systematically.

SE has a long history of formalizing design patterns, i.e., reusable solutions to common design problems. A design pattern provides a high-level template for addressing specific challenges rather than a rigid implementation structure. This concept can be extended to promptware engineering, where recurring prompt structures and strategies can be classified as 'prompt design patterns,' offering systematic frameworks for designing effective prompts.

O5: Formalized prompt design patterns (C1, C2, C3, C6, C7, C10). The SE community can collaborate with AI researchers to formalize existing prompt structures into well-defined design patterns. For example, few-shot prompting parallels the Factory pattern in SE, where contextual examples act as templates for generating outputs. Similarly, CoT prompting resembles the Builder pattern, as intermediate steps are incrementally constructed before producing a final output.

Formalizing these patterns promotes standardization (C1), ensuring consistent practices across the field. It also enables systematic optimization, improving prompt quality and reducing ambiguity (C2, C3). Additionally, standardization can help delineate LLM capability boundaries (C6), clarifying what different prompt structures can and cannot achieve.

Beyond formalizing existing patterns, there is an opportunity to identify new prompt design patterns through systematic experimentation, empirical evaluation, and best practice documentation. For instance, recursive prompting, where the output of one iteration serves as input for the next, could emerge as a distinct pattern for tasks requiring sustained memory tracking (C10). Furthermore, integrating failure-resilient prompt patterns could help mitigate hallucinations and improve fault tolerance (C7).

O6: Prompt design tools (C1). Just as SE provides tools such as UML diagrams to support the implementation of design patterns, the SE community can develop specialized tools for prompt design. These tools could assist prompt engineers in visualizing, assembling, and validating prompt structures (C1), streamlining the development process and enhancing prompt effectiveness.

O7: Prompt design metrics (C1, C4, C5, C7, C10). In traditional SE, design quality is typically assessed using metrics such as cohesion, coupling, and complexity. Applying similar metrics to prompt design could establish a structured and rigorous evaluation framework for prompts. Cohesion could measure how effectively the components of a prompt work together to achieve the desired outcome. Coupling might evaluate the extent to which a prompt depends on external systems, such as retrieval-based components in RAG. Complexity could assess the structural intricacy of a prompt (C1) and the cognitive load it imposes on the LLM (C5).

Additional prompt-specific metrics could further enhance evaluation. A memory evaluation metric could quantify how well a prompt maintains context across multiple turns, ensuring critical information persists throughout an interaction (C10). A probabilistic determinism score could measure output variance for the same input, assessing reliability across different LLM versions and temperature settings (C4). Moreover, error-handling metrics could evaluate how effectively a prompt minimizes hallucinations and ambiguous responses (C7).

O8: Prompt pattern repositories (C3, C6, C9, C10). A shared repository of prompt design patterns, akin to the design pattern catalog in SE, could enhance innovation and standardization in promptware engineering. Such a repository would provide promptware engineers with a collection of well-documented, reusable solutions for common challenges, facilitating faster adaptation and iteration. By reducing reliance on trial-and-error approaches, this resource would promote consistency and improve prompt quality (C3).

Additionally, the repository could include capability-aware prompt patterns, documenting which designs are best suited for specific LLM functionalities (C6).

Security should also be a core component of this repository (C9). A dedicated section on adversarial robustness patterns could catalog techniques for mitigating prompt injection and unintended information leakage.

Finally, patterns for long-term dependency tracking (e.g., summarization prompts that maintain context across multiple exchanges) could help address LLM statelessness, a key limitation in multi-turn interactions (C10).

4.3 Prompt Implementation

In traditional SE, the implementation phase focuses on writing programs that translate specifications into code. In the context of promptware engineering, implementation involves crafting prompts as natural language instructions to be processed by LLMs. Drawing from established SE practices, we highlight several research directions that could enhance the implementation of prompts.

O9: Prompt-centric programming languages (C1, C2, C3, C5, C7). A key opportunity in promptware engineering is the development of prompt-centric programming languages designed to structure, manipulate, and manage prompts systematically. While existing frameworks like LangChain and Liquid prompt templates support chaining and templating, they lack formal programming features such as type systems, static analysis, and error handling (C1, C3, C7). Traditional languages, on the other hand, are not well-suited for handling the nuances of natural language.

A prompt-centric language could bridge this gap by formalizing prompt constructs and contextual dependencies, reducing ambiguity, and improving output consistency (C2). It could also incorporate mechanisms for error detection and handling (C7) and support modular prompt creation (C1), enabling reusable components that integrate seamlessly into larger software systems. By combining human-like interaction with deterministic engineering principles (C5), such a language would enhance the reliability, scalability, and maintainability of promptware.

O10: Prompt compilation (C1, C2, C9). Natural language is ambiguous and context-dependent (C2), making it difficult for LLMs to interpret prompts with precision. This challenge resembles a well-known problem in SE, where high-level programming languages must be transformed into structured, optimized machine code to ensure reliable execution. In traditional computing, compilers address this issue by systematically analyzing, refining, and restructuring code before it reaches the processor. Inspired by this concept, we envision prompt compilation, a process that translates human-written prompts into well-defined, optimized representations that LLMs can process more effectively.

Prompt compilation can involve multiple stages to ensure that a prompt is both information-dense and logically structured. First, the prompt undergoes lexical and syntactic analysis, where it is tokenized and parsed to extract key concepts, tasks, and dependencies. The next step involves creating a prompt intermediate representation, a structured, language-agnostic format that serves as an intermediary between raw prompts and LLM execution. This representation enables systematic analysis and transformation of the prompt, ensuring that it is logically sound and free from ambiguity (C2). After the creation of prompt intermediate representation, semantic optimization refines the structure further (C1), making implicit logic explicit, reorganizing sub-tasks,

and converting natural language instructions into a clear execution plan. To improve efficiency, token compression eliminates redundancies, reformats expressions for clarity, and, when necessary, translates the prompt into a more compact form to reduce computational overhead. Finally, as part of the compilation process, security enhancements introduce constraints and safeguards (C9). These prevent prompt injection attacks and unintended outputs, ensuring that the prompt is not only efficient and logically sound but also secure during execution, much like how compilers ensure that the machine code is robust and free from errors before it runs on the processor.

O11: Prompt-centric IDEs (C2, C3, C4, C7, C10). In traditional SE, Integrated Development Environments (IDEs) facilitate coding, debugging, and testing. A similar opportunity exists in promptware engineering to develop prompt-centric IDEs, which is also discussed in a recent paper [24]. These specialized tools could offer features like real-time feedback on prompt performance, automatic generation of prompt variations, and integration with prompt evaluation metrics (C3). By streamlining prompt creation and refinement, such tools could significantly reduce the time and effort required to implement high-quality prompts.

Given the non-deterministic nature of LLM outputs (C4), the IDE could incorporate features that assess probabilistic consistency across prompt outputs, helping engineers track variability and ensure reproducibility. The IDE could also integrate contextual checks to identify ambiguities in prompts (C2), offering error messages to inform potential issues (C7) and suggestions to make them more explicit and deterministic. Additionally, integration with external memory mechanisms (C10) would help maintain the coherence of prompts over time, especially in long-term interactions

O12: Prompt optimization (C1, C2, C3). Prompt optimization is a significant challenge in promptware engineering. Unlike traditional software code, which benefits from established analysis and optimization tools, prompts are highly context-sensitive (C2), requiring a more nuanced approach to optimization. The outputs of prompts can be influenced by subtle variations in wording (C3), context, and structure, making it difficult to ensure consistency and effectiveness across a broad range of tasks. In prompt optimization, factors such as structure, tone, contextual information, and task-specific instructions must be fine-tuned to improve outputs (C1, C2).

To address these challenges, search-based software engineering (SBSE) methodologies [22] offer promising solutions. SBSE techniques, such as genetic algorithms, can automatically explore and evaluate different prompt configurations to identify the optimal combination of elements (such as wording, structure, and context) that maximizes effectiveness across various tasks. By iterating through diverse variations, these methods enhance robustness and mitigate the impact of subtle prompt differences. Additionally, context-aware optimization algorithms can dynamically adjust the prompt elements in real-time, ensuring that the prompt remains effective even as context or task requirements change. Furthermore, multi-objective optimization frameworks can help balance competing factors such as accuracy, consistency, and efficiency, providing a comprehensive solution for prompt optimization.

O13: Online prompt implementation (C2, C6, C10). Prompt development frequently requires online implementation, where responses are generated in real time with minimal delay to ensure prompt user feedback [8]. The need for online prompts stems from the dynamic runtime environment (C6), which contrasts with the more rigid, controlled setup of traditional offline systems. While online systems are preferred for their responsiveness, the transition from offline to online frameworks is not always feasible due to the continuous adaptability required in real-time contexts.

The primary challenge of online prompt implementation lies in effectively balancing real-time adaptability with contextual coherence and response accuracy. Unlike offline systems that rely on static prompts, online systems must continuously evolve in response to user input, system behavior, and fluctuating environmental factors. This inherent dynamism introduces difficulties in maintaining the relevance and consistency of prompts, particularly when handling large-scale or

complex interactions. This area presents significant research opportunities, especially in developing algorithms capable of learning from real-time interactions. Key avenues of exploration include adaptive prompt generation, improving context-aware language understanding (C2), and advancing long-term memory mechanisms that allow models to retain and recall context across multiple sessions (C10).

O14: Role-playing in prompts (C5, C6). Role-playing is a widely-adopted prompting strategy for enhancing the utility of LLMs by simulating real-world roles [41]. A key challenge in role-playing prompts is specifying optimal roles, which requires a deep understanding of the LLM's human-like characteristics and its capabilities in role-playing (C5). This understanding is crucial for ensuring better contextual understanding and engagement. Defining these roles also demands a balance between clarity and flexibility—too rigid a role can limit the model's responses, while too vague a role can lead to loss of context. Additionally, the role must be tailored to the LLM's capabilities (C6), ensuring that it can effectively handle complex and dynamic interactions.

Specifying social roles in prompts can also introduce biases linked to stereotypes present in training data [28]. This raises a significant issue: how can we use role-playing to improve interaction while minimizing bias? Solutions could include refining role definitions, applying bias-mitigation techniques, and continuously evaluating prompts across diverse demographic and cultural contexts. Ethical guidelines and fairness-aware mechanisms should be integrated to ensure role-playing adds value without reinforcing harmful stereotypes.

O15: Prompt libraries and APIs (C1, C2, C9). In SE, reusable code libraries have significantly enhanced development efficiency. A similar approach in promptware engineering could involve curated libraries of reusable prompt templates for common tasks such as summarization, question answering, and translation. These libraries would provide baseline prompts that engineers could easily adapt to suit specific needs. To ensure standardization, these libraries would need to follow consistent structures (C1) and allow for easy customization while minimizing ambiguity (C2).

However, similar to the open-source software ecosystem in SE, prompt engineering must address challenges related to copyright and licensing. Recent research [37, 47] has highlighted the need for frameworks that can define, detect, and resolve intellectual property issues. Developing licensing models that strike a balance between open access, commercial use, and ethical considerations will be crucial for the sustainable growth of promptware engineering.

Standardized prompt APIs for interacting with prompts and LLMs could further streamline integration into larger software ecosystems. These APIs could include methods for prompt assembly and output validation, enabling modular, maintainable, and scalable prompt-based applications. To ensure the security of prompt execution and protect sensitive data, these APIs should incorporate access control mechanisms (C9) that define who can modify or access specific prompt configurations. Additionally, these standardized APIs could support dependency tracking across reusable prompts, improving both the modularity and traceability of the promptware engineering process.

4.4 Prompt Testing and Debugging

Software testing verifies whether software behaves as expected, while debugging identifies and resolves issues. In promptware engineering, prompt testing and debugging detect and correct undesired behaviors triggered by prompts. Unlike traditional testing and debugging, which focus on deterministic code, prompt testing and debugging must handle non-deterministic outputs and ambiguous specifications, introducing unique challenges.

O16: Flaky test of prompts (C2, C4, C5). In SE, a flaky test refers to a test that produces inconsistent results, yielding failures or successes unpredictably across multiple runs without changes to the underlying code [32]. This issue is particularly prevalent in prompt testing, where

the probabilistic nature of natural language (C2), combined with the non-deterministic outputs of LLMs (C4), creates challenges in ensuring consistent test results.

To determine whether a test passes or fails, multiple attempts may be necessary, relying on aggregated or consensus-based assessments to distinguish between random fluctuations and genuine prompt flaws. One potential solution is to move beyond the traditional binary pass/fail approach, introducing a success threshold (e.g., requiring outputs to be highly similar in at least 80% of test runs). While this reduces the impact of random variations, it does not entirely eliminate the uncertainty associated with flaky tests.

To mitigate randomness, researchers often adjust the temperature parameter, which influences the level of creativity and variability in the generated text [35]. A temperature of zero minimizes randomness, yielding more deterministic outputs. However, even with this setting, LLMs still exhibit some degree of variability [35]. Moreover, real-world applications typically require higher temperature settings to encourage more dynamic and creative responses (C5). Testing with a temperature of zero may not fully represent real-world scenarios, leaving flaky test in prompt testing as a significant challenge. Flaky test methods for prompts remains an open problem, and further research is needed to develop more dependable solutions.

O17: Test input generation in prompt testing (C1, C4, C5, C6). Test input for prompt testing refers to the specific values assigned to different elements (i.e., variables) within a prompt. In practice, prompts may receive inputs from both LLMs and other software components. For example, in a code generation scenario, a prompt might include a task description provided by other components, as well as feedback or contextual information from prior LLM-generated outputs. This interaction introduces challenges for test input generation in prompt testing.

Specifically, it becomes difficult to generate representative and comprehensive test data that covers the range of possible inputs the prompt might encounter during actual usage (C1). The dynamic nature of inputs, such as changing task descriptions or evolving context from prior LLM outputs (C6), means that test inputs must account for a wide variety of conditions. Additionally, the interdependence between prompts and external software components adds another layer of complexity, as testing requires ensuring that both the prompt and the surrounding system behave cohesively and predictably.

Further complicating matters, the variability introduced by previous LLM outputs (which may differ slightly each time, due to factors like randomness) makes it hard to ensure consistent test coverage (C4). Test inputs need to be designed to capture this stochastic behavior while still allowing for reliable evaluation of the prompt's functionality. This dynamic and multifaceted nature of prompt testing demands robust input generation strategies that can handle the unpredictability of real-world interactions.

O18: Test oracle in prompt testing (C1, C2, C3, C4). The test oracle problem, which involves determining the correct behavior in response to an input, is a well-known challenge in software testing [9]. In prompt testing, a test oracle is a mechanism or reference that assesses whether the output triggered by a prompt with a given test input is correct or acceptable. Unlike traditional testing, where the expected output is typically clear and deterministic, prompt testing faces a greater challenge due to the inherent subjectivity and open-ended nature of many LLM prompts (C1, C2). For example, prompts might ask the model to provide helpful or polite responses or elicit creative problem-solving. These specifications are often vague and context-dependent, complicating the determination of a 'correct' output (C3).

In current promptware practices, manual evaluation is often used to assess output correctness [34]. While effective, this method is time-consuming and susceptible to human biases. Techniques like multi-reviewer consensus or double-blind evaluations can mitigate these biases by offering more balanced judgments. An alternative, the LLM-as-a-judge approach, uses one LLM to evaluate

another's output [13]. While this method can expedite testing by filtering out clearly incorrect or inappropriate responses, it may amplify biases if the models share similar training data or limitations. Therefore, human oversight remains essential, particularly in high-stakes contexts.

Metamorphic testing is a common technique used to address the traditional test oracle problem. This method defines relationships, known as metamorphic relations, between variations in input and expected output consistency [14]. For example, when testing a prompt for code generation, rephrasing the task description should result in consistent model responses if the prompt is well-constructed. If significant discrepancies arise between the rephrased inputs and outputs, it may indicate a flaw in the prompt. However, because LLMs often exhibit non-deterministic behavior—producing different outputs for the same prompt (C4), it is difficult to assess whether the variation is due to prompt inconsistencies or model behavior. To address this, research can focus on developing metamorphic testing techniques specifically designed to test how small, controlled changes in input influence outputs, ensuring that expected behavior remains consistent across variations.

O19: Test adequacy in prompt testing (C1). Test adequacy is a key concept in traditional software testing, used to assess the coverage provided by existing tests [20]. In this context, it measures how thoroughly various aspects of the software, such as different code paths or functional areas, are tested. In prompt testing, adequacy shifts focus to evaluating how well different components of a prompt, such as task descriptions, context, examples, and formatting requirements, are covered. Unlike traditional testing, which often relies on metrics like branch coverage, these methods are not directly applicable to prompt testing due to the unstructured and flexible nature of natural language. One way to improve test adequacy is by developing formalized structures (C1) for prompt components, which would provide a clearer basis for establishing coverage metrics.

O20: Unit testing and integration testing of prompts (C4, C8). Unit testing involves evaluating individual prompts or closely related sets of prompts in isolation to ensure that each prompt functions as intended. This approach verifies that each prompt meets its specific requirements without interference from other prompts or external contexts. However, in complex LLM-based softwre, prompts are often chained together [30], as in conversation flows where each response depends on previous inputs. Integration testing ensures that these interconnected prompts produce coherent, accurate, and contextually consistent outputs when used together. Errors that may go undetected during unit testing can surface only when prompts interact in real-world conditions, highlighting the importance of integration testing to verify the system's overall behavior. Since determinism and execution control are often difficult to guarantee in LLMs (C4, C8), enhancing integration testing might involve developing tools that give developers more control over the flow of execution, including debugging tools that make internal processes more observable. This will help identify where unexpected results emerge during prompt interactions.

O21: Non-functional testing (C5, C9): As LLM-based systems are increasingly adopted in human-centric applications, ensuring that non-functional requirements such as fairness, security, and privacy are met is becoming critically important. Non-functional testing helps verify that these aspects are addressed thoroughly.

Fairness testing. Since LLMs exhibit human-like characteristics (C5), they may also display human-like social biases or discriminatory tendencies when prompted with inappropriate inputs [28, 42]. These biases can lead to the unfair treatment of specific groups or perpetuate harmful stereotypes related to various roles. If left unchecked, the repeated use of biased responses can normalize these stereotypes, subtly influencing public perception and reinforcing social inequalities. Fairness testing aims to identify biases caused by prompts.

Security testing. Security testing aims to identify vulnerabilities that could compromise the safety and integrity of LLM-based systems. For example, adversarial attacks like prompt injection (C9) exploit weaknesses by crafting malicious inputs to manipulate the LLM or bypass safeguards [16].

Privacy testing. Given that LLM-based software often handles sensitive information, ensuring privacy is crucial. Prompts must undergo rigorous testing to ensure they do not inadvertently expose private or confidential data (C9). For example, if a prompt prompts an LLM to disclose sensitive information, whether from user input or internal training data, it constitutes a serious privacy breach.

O22: Prompt debugging (C4, C6, C7, C8, C10). Poorly constructed or ambiguous prompts may result in incorrect or undesirable outputs. Prompt debugging focuses on identifying and resolving these issues.

A primary challenge in prompt debugging is reproducing bugs, as LLMs often exhibit flakiness, i.e., identical prompts can produce inconsistent outputs (C4). This variability complicates bug reproduction, especially when context, recent interactions, or model updates change outputs. To address this, robust debugging mechanisms are essential. Tools that capture and replay the system's state during prompt execution, including model configurations, input context, and reproducibility factors like temperature settings, can help developers reproduce bugs and analyze discrepancies across different configurations.

Once a bug is reproduced, identifying its root cause, i.e., whether in the prompt design or the model's behavior, becomes crucial. The black-box nature of LLMs (C6) complicates this, as developers lack visibility into the model's internal processes. Additionally, the absence of execution control (C8) means developers cannot inspect intermediate states. As a result, prompt ebugging often relies on indirect techniques like prompt decomposition and ablation studies, modifying parts of the prompt to isolate the cause. However, the inherent flakiness of LLMs (C4) makes localization difficult, as the same prompt may yield different results across runs.

After localization, the next challenge is bug fixing, where the problematic section of the prompt is adjusted while preserving its intended functionality. While bug fixes may resolve one issue, they can inadvertently introduce new ones, especially in multi-step or CoT prompts. To mitigate this, automated tools should be implemented to suggest and verify prompt modifications based on known patterns of failure. This pattern-based approach would help developers apply proven solutions to recurring problems, thereby streamlining the debugging process and reducing errors.

Additionally, incorporating error-handling mechanisms within LLMs could enhance their ability to provide explicit feedback when an issue arises. Instead of leaving developers to infer the problem, LLMs could flag uncertainties or ambiguities in the prompt and suggest improvements. This would address the lack of structured error messages (C7) and make the identification and resolution of issues more efficient.

Finally, for complex, multi-step prompts, maintaining continuity across interactions is crucial. Since LLMs lack persistent memory (C10), external mechanisms for contextual memory could be used to maintain coherence across multiple exchanges. This would help reduce errors caused by the loss of context and ensure smoother debugging, particularly for longer chains of reasoning.

4.5 Prompt Evolution

Software evolution typically involves iterative development and updates. Similarly, prompts require proactive adaptation and continuous refinement, especially in dynamic environments.

O23: Prompt evolution driven by code, LLM, and user feedback (C1, C4, C5, C6). Prompt evolution in LLM-based software is influenced by code changes, LLM updates, and user feedback. Unlike programming languages, prompts lack explicit syntax rules, making adaptation more challenging (C1). Additionally, LLMs generate responses probabilistically, complicating consistent

behavior from prompt modifications (C4). As LLMs evolve with updates and new training data, prompts must be revalidated to prevent performance degradation (C6). LLMs' human-like reasoning (C5) also introduces susceptibility to implicit biases and shifting linguistic patterns, requiring tracking and adaptation.

To streamline prompt evolution, researchers and practitioners can develop context-aware prompt management systems that automatically detect shifts in code, model behavior, and user feedback, adjusting prompts accordingly. One approach is to introduce structured prompt templates and metadata tracking to improve control over prompt modifications. Additionally, automated validation mechanisms can assess whether prompt adjustments yield reliable results.

O24: Versioning and traceability (C2, C3, C4, C6, C8, C9). In traditional SE, version control systems like Git are essential for tracking code changes and maintaining traceability. Similarly, promptware engineering would benefit from specialized version control tools to track prompt iterations, document modifications, and ensure accountability (C3). Such systems would enable engineers to compare prompt versions, ensuring consistent refinement and structured evolution.

Since prompts interact dynamically with evolving LLMs (C6), maintaining compatibility between prompt versions and software upgrades is critical. Any modifications, especially those prompted by LLM updates, code changes, user feedback, or shifting use cases, should be thoroughly documented. Given the non-deterministic nature of LLM execution (C4), precise versioning strategies are necessary to mitigate unintended variations in model behavior.

To support these, prompt versioning systems should incorporate detailed changelogs, update metadata, and explicit compatibility requirements, mirroring the principles of semantic versioning in traditional software (C2). Additionally, automated diff-checking mechanisms could highlight changes in prompts and their impact on LLM responses, improving reliability.

Furthermore, access control mechanisms should be integrated into prompt management systems (C9) to ensure that modifications are authorized and traceable, preventing unauthorized edits or security vulnerabilities. Since debugging prompts remains experimental and lacks structured debugging tools (C8), versioning should support rollback mechanisms to restore previously stable versions when regressions occur.

5 Conclusion

In this paper, we highlight 10 unique characteristics of promptware, which arise from its natural language programming and the use of LLMs as a runtime, in contrast to traditional software paradigms. These characteristics introduce specific challenges that current experimental, trial-and-error practices fail to adequately address. To overcome these limitations, we propose promptware engineering, a systematic methodology that integrates established software engineering principles into prompt creation and optimization, moving beyond the ad-hoc methods currently in use. To support this vision, we present a roadmap with 24 research opportunities across critical areas such as prompt requirements engineering, design, implementation, testing, debugging, and evolution.

References

[1] 2025. Enhance results with prompt engineering strategies. https://platform.openai.com/docs/guides/prompt-engineering.
[2] 2025. Introducing Claude. https://www.anthropic.com/news/introducing-claude.
[3] 2025. LangChain prompt templates. https://python.langchain.com/docs/concepts/prompt_templates/.
[4] 2025. Liquid prompt's documentation. https://liquidprompt.readthedocs.io/en/stable/.
[5] 2025. Prompt engineering best practices for ChatGPT. https://help.openai.com/en/articles/10032626-prompt-engineering-best-practices-for-chatgpt.
[6] 2025. Prompting. https://www.llama.com/docs/how-to-guides/prompting/.

[7] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. 2023. GPT-4 technical report. *arXiv preprint arXiv:2303.08774* (2023).

[8] Nadia Alshahwan, Mark Harman, Inna Harper, Alexandru Marginean, Shubho Sengupta, and Eddy Wang. 2024. Assured LLM-based software engineering. In *Proceedings of the 2nd IEEE/ACM International Workshop on Interpretability, Robustness, and Benchmarking in Neural Software Engineering, InteNSE@ICSE 2024*. 7–12.

[9] Earl T. Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. 2015. The oracle problem in software testing: A survey. *IEEE Transactions on Software Engineering* 41, 5 (2015), 507–525.

[10] Xiao Bi, Deli Chen, Guanting Chen, Shanhuang Chen, Damai Dai, Chengqi Deng, Honghui Ding, Kai Dong, Qiushi Du, Zhe Fu, et al. 2024. Deepseek LLM: Scaling open-source language models with longtermism. *arXiv preprint arXiv:2401.02954* (2024).

[11] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language models are few-shot learners. In *Proceedings of Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020*.

[12] Yupeng Chang, Xu Wang, Jindong Wang, Yuan Wu, Linyi Yang, Kaijie Zhu, Hao Chen, Xiaoyuan Yi, Cunxiang Wang, Yidong Wang, et al. 2024. A survey on evaluation of large language models. *ACM Transactions on Intelligent Systems and Technology* 15, 3 (2024), 1–45.

[13] Guiming Hardy Chen, Shunian Chen, Ziche Liu, Feng Jiang, and Benyou Wang. 2024. Humans or LLMs as the judge? A study on judgement bias. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing, EMNLP 2024*, Yaser Al-Onaizan, Mohit Bansal, and Yun-Nung Chen (Eds.). 8301–8327.

[14] Tsong Yueh Chen, Fei-Ching Kuo, Huai Liu, Pak-Lok Poon, Dave Towey, T. H. Tse, and Zhi Quan Zhou. 2018. Metamorphic testing: A review of challenges and opportunities. *Comput. Surveys* 51, 1 (2018), 4:1–4:27.

[15] Gabriele De Vito. 2024. Assessing healthcare software built using IoT and LLM technologies. In *Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering*. 476–481.

[16] Edoardo Debenedetti, Jie Zhang, Mislav Balunovic, Luca Beurer-Kellner, Marc Fischer, and Florian Tramèr. 2024. AgentDojo: A dynamic environment to evaluate prompt injection attacks and defenses for LLM agents. In *Proceedings of the Thirty-eight Conference on Neural Information Processing, NeurIPS 2024, Systems Datasets and Benchmarks Track*.

[17] Mateusz Dolata, Norbert Lange, and Gerhard Schwabe. 2024. Development in times of hype: How freelancers explore Generative AI?. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering, ICSE 2024*. 1–13.

[18] Angela Fan, Beliz Gokkaya, Mark Harman, Mitya Lyubarskiy, Shubho Sengupta, Shin Yoo, and Jie M. Zhang. 2023. Large language models for software engineering: Survey and open problems. In *Proceedings of IEEE/ACM International Conference on Software Engineering: Future of Software Engineering, ICSE-FoSE 2023*. 31–53.

[19] Tyler Giallanza and Declan Iain Campbell. 2024. Context-sensitive semantic reasoning in large language models. In *ICLR 2024 Workshop on Representational Alignment*.

[20] Milos Gligoric, Alex Groce, Chaoqiang Zhang, Rohan Sharma, Mohammad Amin Alipour, and Darko Marinov. 2013. Comparing non-adequate test suites using coverage criteria. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis, ISSTA 2013*. 302–313.

[21] Yaoqi Guo, Zhenpeng Chen, Jie M. Zhang, Yang Liu, and Yun Ma. 2024. Personality-guided code generation using large language models. *CoRR* abs/2411.00006 (2024).

[22] Mark Harman and Bryan F Jones. 2001. Search-based software engineering. *Information and software Technology* 43, 14 (2001), 833–839.

[23] Ahmed E. Hassan, Dayi Lin, Gopi Krishnan Rajbahadur, Keheliya Gallaba, Filipe Roseiro Côgo, Boyuan Chen, Haoxiang Zhang, Kishanthan Thangarajah, Gustavo Ansaldi Oliva, Jiahuei (Justina) Lin, Wali Mohammad Abdullah, and Zhen Ming (Jack) Jiang. 2024. Rethinking software engineering in the era of foundation models: A curated catalogue of challenges in the development of trustworthy FMware. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering, FSE 2024*. 294–305.

[24] Ahmed E Hassan, Gustavo A Oliva, Dayi Lin, Boyuan Chen, and Zhen Ming Jiang. 2024. Rethinking software engineering in the foundation model era: From task-driven AI Copilots to goal-driven AI pair programmers. *arXiv preprint arXiv:2404.10225* (2024).

[25] Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang. 2024. Large language models for software engineering: A systematic literature review. *ACM Transactions on Software Engineering and Methodology* 33, 8 (2024), 1–79.

[26] Jen-tse Huang, Man Ho Lam, Eric John Li, Shujie Ren, Wenxuan Wang, Wenxiang Jiao, Zhaopeng Tu, and Michael Lyu. 2024. Apathetic or empathetic? Evaluating LLMs' emotional alignments with humans. In *Proceedings of the*

Thirty-eighth Annual Conference on Neural Information Processing Systems.
[27] Patrick S. H. Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. 2020. Retrieval-augmented generation for knowledge-intensive NLP tasks. In *Proceedings of Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020*.
[28] Xinyue Li, Zhenpeng Chen, Jie M. Zhang, Yiling Lou, Tianlin Li, Weisong Sun, Yang Liu, and Xuanzhe Liu. 2024. Benchmarking bias in large language models during role-playing. *CoRR* abs/2411.00585 (2024).
[29] Yinheng Li, Shaofei Wang, Han Ding, and Hang Chen. 2023. Large language models in finance: A survey. In *Proceedings of the fourth ACM international conference on AI in finance*. 374–382.
[30] Jenny T. Liang, Melissa Lin, Nikitha Rao, and Brad A. Myers. 2024. Prompts are programs too! Understanding how developers build software containing prompts. *CoRR* abs/2409.12447 (2024).
[31] Junwei Liu, Kaixin Wang, Yixuan Chen, Xin Peng, Zhenpeng Chen, Lingming Zhang, and Yiling Lou. 2024. Large language model-based agents for software engineering: A survey. *arXiv preprint arXiv:2409.02977* (2024).
[32] Qingzhou Luo, Farah Hariri, Lamyaa Eloussi, and Darko Marinov. 2014. An empirical analysis of flaky tests. In *Proceedings of the 22nd ACM SIGSOFT international symposium on foundations of software engineering, FSE 2014*. 643–653.
[33] Alina Mailach, Sebastian Simon, Johannes Dorn, and Norbert Siegmund. 2025. Practitioners' discussions on building LLM-based applications for production. In *Proceedings of the IEEE/ACM 4th International Conference on AI Engineering - Software Engineering for AI, CAIN 2025*.
[34] Nadia Nahar, Christian Kästner, Jenna Butler, Chris Parnin, Thomas Zimmermann, and Christian Bird. 2024. Beyond the comfort zone: Emerging solutions to overcome challenges in integrating LLMs into software products. *arXiv preprint arXiv:2410.12071* (2024).
[35] Shuyin Ouyang, Jie M Zhang, Mark Harman, and Meng Wang. 2024. An empirical study of the non-determinism of ChatGPT in code generation. *ACM Transactions on Software Engineering and Methodology* (2024).
[36] Chris Parnin, Gustavo Soares, Rahul Pandita, Sumit Gulwani, Jessica Rich, and Austin Z Henley. 2023. Building your own product Copilot: Challenges, opportunities, and needs. *arXiv preprint arXiv:2312.14231* (2023).
[37] Huali Ren, Anli Yan, Chong-zhi Gao, Hongyang Yan, Zhenxin Zhang, and Jin Li. 2024. Are you copying my prompt? Protecting the copyright of vision prompt for VPaaS via watermark. *CoRR* abs/2405.15161 (2024).
[38] Pranab Sahoo, Ayush Kumar Singh, Sriparna Saha, Vinija Jain, Samrat Mondal, and Aman Chadha. 2024. A systematic survey of prompt engineering in large language models: Techniques and applications. *arXiv preprint arXiv:2402.07927* (2024).
[39] Mahan Tafreshipour, Aaron Imani, Eric Huang, Eduardo Almeida, Thomas Zimmermann, and Iftekhar Ahmed. 2025. Prompting in the wild: An empirical study of prompt evolution in software repositories. In *Proceedings of the 22nd IEEE/ACM International Conference on Mining Software Repositories, MSR 2025*.
[40] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. 2023. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971* (2023).
[41] Yu-Min Tseng, Yu-Chao Huang, Teng-Yun Hsiao, Yu-Ching Hsu, Jia-Yin Foo, Chao-Wei Huang, and Yun-Nung Chen. 2024. Two tales of persona in llms: A survey of role-playing and personalization. *arXiv preprint arXiv:2406.01171* (2024).
[42] Yuxuan Wan, Wenxuan Wang, Pinjia He, Jiazhen Gu, Haonan Bai, and Michael R. Lyu. 2023. BiasAsker: Measuring the bias in conversational AI system. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2023*. 515–527.
[43] Junjie Wang, Yuchao Huang, Chunyang Chen, Zhe Liu, Song Wang, and Qing Wang. 2024. Software testing with large language models: Survey, landscape, and vision. *IEEE Transactions on Software Engineering* (2024).
[44] Shen Wang, Tianlong Xu, Hang Li, Chaoli Zhang, Joleen Liang, Jiliang Tang, Philip S Yu, and Qingsong Wen. 2024. Large language models for education: A survey and outlook. *arXiv preprint arXiv:2403.18105* (2024).
[45] Wei Wang, Vincent W Zheng, Han Yu, and Chunyan Miao. 2019. A survey of zero-shot learning: Settings, methods, and applications. *ACM Transactions on Intelligent Systems and Technology (TIST)* 10, 2 (2019), 1–37.
[46] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. 2022. Chain-of-thought prompting elicits reasoning in large language models. In *Proceedings of Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022*.
[47] Yong Yang, Xuhong Zhang, Yi Jiang, Xi Chen, Haoyu Wang, Shouling Ji, and Zonghui Wang. 2024. PRSA: Prompt reverse stealing attacks against large language models. *CoRR* abs/2402.19200 (2024).

Received 2025; revised 2025; accepted 2025