

Prompts Blend Requirements and Solutions: From Intent to Implementation

Shalini Chakraborty ✉ 

University of Bayreuth, Germany

Jan-Philipp Steghöfer ✉ 

XITASO GmbH IT and Software Solutions, Augsburg, Germany

Abstract

AI coding assistants are fundamentally reshaping software development by shifting developers' effort from writing code toward specifying intent through natural language prompts. In emerging chat-based development practices such as *vibe coding*, prompts mediate the transformation of human intent into executable software. While Requirements Engineering (RE) emphasizes capturing, validating, and evolving requirements, current prompting practices remain informal and ad hoc. In this vision paper, we argue that prompts represent lightweight, evolving requirements artifacts that combine expressions of user needs with varying degrees of solution guidance. We use an existing conceptual model that decomposes prompts into three interrelated dimensions: *Functionality and Quality* (capturing intended system requirements), *General Solutions* (capturing architectural strategies and technology choices), and *Specific Solutions* (capturing implementation-level constraints and directives). Building on this conceptualization, we formulate four research hypotheses concerning (i) the evolution of prompts over time, (ii) the influence of user characteristics on prompt evolution, (iii) the relationship between prompt content and requirements validation and verification activities, and (iv) the impact of prompt characteristics on requirements and resulting software quality. We envision an empirical research agenda combining real-world AI-assisted development data, corpus analysis, and controlled experimentation to investigate these hypotheses and derive evidence-based practices for requirements-aware prompt engineering. By reframing prompts through the lens of RE, we position prompting not merely as an interaction mechanism with AI systems, but as a central software engineering concern requiring systematic study.

2012 ACM Subject Classification Software and its engineering → Software development techniques; Software and its engineering → Requirements analysis; Software and its engineering → Development frameworks and environments; Social and professional topics → Socio-technical systems

Keywords and phrases Requirements Engineering, Vibe Coding, Prompts, AI

Digital Object Identifier 10.4230/LIPIcs.ESEM.2026.71

Category Emerging Results, Vision & Reflection Track Paper

Supplementary Material *Software*: <https://zenodo.org/records/20329289>

Acknowledgements Perplexity.ai and Claude Sonnet 4.5 and 4.6 were used to critique drafts of the manuscript, refine hypotheses, and elicit feedback on different aspects of the conceptual model. Claude Sonnet 4.6 was used to generate the scripts we used to prepare the dataset, to classify information in prompts into the three areas of the triangle, and to generate the corresponding tables. In addition, it created some of the text passages in this work.



© Shalini Chakraborty and Jan-Philipp Steghöfer;
licensed under Creative Commons License CC-BY 4.0

20th International Symposium on Empirical Software Engineering and Measurement (ESEM 2026).

Editors: Robert Feldt, Maria Paasivaara, Daniel Mendez, Stefan Wagner, and Marvin Muñoz Barón;

Article No. 71; pp. 71:1–71:14



Leibniz International Proceedings in Informatics

Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

Software engineering has been reshaped by AI-assisted chat-based coding systems. Tools like GitHub Copilot¹, Tabnine², and Codex³ enable developers to describe desired functionality, constraints, or behavior in natural language rather than writing code; the AI then responds with implementations, design suggestions, or complete components. The quality of these prompts—their clarity, completeness, and structure—often determines whether the resulting code meets expectations. Consequently, prompt engineering and requirements engineering now overlap more than in traditional workflows: developers must think both about *what* must be built (requirements) and *how* to communicate it to the model (prompt). In emerging paradigms like vibe coding [23] and agentic coding [35, 25], natural language prompts become the primary artifact bridging human intent and executable software. Unlike traditional workflows that maintain clear divisions between requirements, design, and implementation, chat-based development collapses these boundaries: a single prompt encapsulates requirements, architectural strategy, and implementation constraints simultaneously.

Yet prompts remain informal, ephemeral, and ad hoc. They lack the rigor traditionally associated with Requirements Engineering (RE). We argue that prompts should be understood as lightweight, evolving requirement artifacts that blend requirements with solution guidance, and that systematic prompt engineering represents a new frontier for RE in the age of generative AI.

This paper builds on the existing prompt triangle [6] as an emerging result, a conceptual model decomposing prompts into three interrelated dimensions: (1) Functionality and Quality (the requirement), (2) General Solutions (architectural strategy and technology choices), and (3) Specific Solutions (implementation-level constraints). We validate this model using the DevGPT dataset [37], position it relative to existing prompt patterns and RE validation techniques, and formulate four testable hypotheses about prompt evolution, user characteristics, requirements verification in iterative prompting, and the relationship between prompt refinement and code quality. Our vision is to establish empirically-informed best practices for requirements-aware prompt engineering, rejuvenating RE for AI-assisted development.

2 Chat-based Coding Assistants

AI coding assistants operate in two distinct modes that shape developer interaction fundamentally differently. *Code completion interfaces* work within existing code structures, assisting with boilerplate, method bodies, and idiomatic patterns where developers provide the architectural skeleton. *Chat-based interfaces* enable developers to describe functionality, constraints, and solution strategies in natural language, potentially revamping entire codebases from a single prompt. There is a rapidly growing body of literature on AI coding assistants, with new papers posted to arXiv and other repositories without peer review daily. The peer-reviewed sources are still few and far between and often make no distinctions between chat-based interfaces to the AI and code completions. However, it is crucial to make this distinction since they differ significantly in terms of user experience: while code completion interfaces work within existing code and help the developer implement certain

¹ <https://azure.microsoft.com/en-us/products/github/copilot>

² https://www.tabnine.com/?utm_source=chatgpt.com

³ https://openai.com/codex/?utm_source=chatgpt.com

boilerplate code, smaller features, or tests, chat interfaces have a much bigger scope and give the developer the chance to revamp entire code bases with a single prompt. We are interested in the former setting. Sergeyuk et al. [29], for instance, conducted a study with 481 respondents in which they asked about tasks such as “Implementing new features”, “Writing tests”, or “Bug triaging”. Depending on whether a code completion interface or a chat interface is used, the user experience will be quite different. For a code completion interface, the developer will create a structure for the feature with functions etc. that might have small pieces of documentation that describes the desired functionality. For a chat interface, the developer might write a single prompt and include elements of the requirements and the solution approach as we argue above. This distinction is methodologically critical [2, 32, 29]: code completion preserves traditional incremental programming, while chat-based prompting externalizes problem-solving into conversational exchanges where a single prompt conflates requirements specification, design rationale, validation-verification criteria, and implementation intent. These are a methodically and cognitively distinct approaches that should not be treated the same when asking developers how much they “enjoy” these activities. Conflating the two modes risks masking key differences in cognitive effort, developer mental models, and the role of requirements [36]. This paper focuses on **chat-based interfaces** and their implications for requirements engineering.

3 Prompts and Requirements

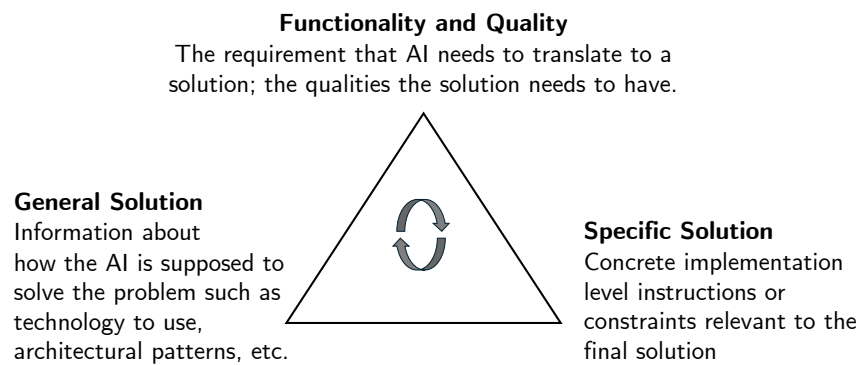
Prompt engineering refers to the practice of carefully crafting and refining prompts to guide large language models (LLMs) toward producing desired outputs [24, 8]. In the context of software development, prompt engineering is increasingly seen as a crucial skill for effectively leveraging chat-based coding assistants. However, while prompts may contain elements of system requirements or design intent, they are often ad hoc and lack the structure of formal requirements [2]. Many developers struggle to provide the right level of detail and to translate requirements into prompts consistently [33]. This raises an important question for RE: **How can the rigor of RE be infused into prompting practices?** Viewing prompts as lightweight but evolving requirement artifacts offers a way to bridge the gap between informal developer intentions and structured requirements specification [34]. Such an integration will enhance the reliability of AI-generated solutions and redefine RE practices in the era of AI-assisted chat-based development.

3.1 A Conceptual Model of Prompts

Recent work has begun to examine the role of prompts in software development and RE [27, 14]. However, understanding prompts as artifacts within AI-assisted development remains an open challenge. This is particularly relevant in emerging practices such as *vibe coding*, where software is iteratively developed through conversational interactions with AI [20], and *agentic coding*, where autonomous AI agents reason about and execute development tasks, guided by the developer and a large language model (LLM). In such settings, prompts become the primary medium through which developer intent is communicated and translated into executable systems. Consequently, prompts increasingly occupy a role that extends beyond simple instructions and intersects with core concerns of RE, such as requirements elicitation, specification, and evolution.

To better understand this role, we draw on an existing conceptual model that decomposes prompts into three interrelated components. The *Prompt Triangle* (Figure 1) provides a practical and conceptual lens for structuring prompts and assessing their completeness from an RE perspective. We posit that most prompts used in chat-based coding contain these three dimensions implicitly.

71:4 The Prompt Triangle



■ **Figure 1** Prompt Triangle: prompts consist of three interrelated dimensions: Functionality and Quality, General Solutions, and Specific Solutions. The circular relationship indicates iterative refinement across development interactions.

The *Prompt Triangle* consists of three interrelated dimensions. *Functionality and Quality* captures intended system behavior and desired quality characteristics, i.e., what the system should do and how well it should perform, encompassing both functional requirements and quality requirements such as performance, security, accessibility, reliability, and usability. *General Solutions* capture higher-level guidance on how functionality should be realized, including architectural decisions, preferred technologies, design paradigms, and development strategies. *Specific Solutions* represent detailed implementation instructions and constraints that further narrow the solution space and guide concrete realization. Embedding quality attributes directly within the functionality description ensures that prompts capture not only *what* a system should achieve, but also *how well* it should achieve it. Whenever possible, these requirements should be expressed in a measurable way, while guidance on realization mechanisms remains within the General and Specific Solutions dimensions. Co-locating quality requirements with functionality stabilizes developer intent and creates a clearer foundation for validation and verification activities.

Unlike traditional requirements artifacts, which often separate requirements from implementation concerns, prompts frequently blend these dimensions within a single interaction. During iterative development, prompts evolve by expressing intent, refining solution strategies, and constraining implementation details. The circular relationship within the Prompt Triangle represents this continuous refinement process, where movement across components occurs as developers and AI systems progressively converge toward a desired solution.

As an **illustrative example** of the Prompt Triangle, consider the following prompt:

Build a React web app that shows the current weather for a given city, and make it fast, reliable, safe, and accessible. Use server-side rendering; fetch data from OpenWeather. Style the page cleanly and show city, condition, humidity, wind, and Celsius temperature with an icon. It should load quickly on a typical phone, show a loading state, retry if the network fails, and display clear errors. Keep the API key on the server, avoid logging personal details, and ensure people can use the app with a keyboard and screen readers with readable colors.

Although the prompt appears as a single instruction, it combines multiple concerns that can be mapped to the dimensions of the Prompt Triangle. *Functionality and Quality* includes the requirement to build a weather application, together with quality characteristics such as performance, reliability, security, and accessibility (e.g., fast loading, robust error handling, privacy-preserving logging, and support for keyboard and screen-reader interactions). *General Solutions* guide broader implementation decisions, including the use of React, server-

side rendering, and integration with the OpenWeather API. Finally, *Specific Solutions* further constrain implementation through concrete design choices such as displaying weather attributes (city, humidity, wind, temperature, icons), applying a clean visual style, and ensuring that API keys remain securely stored on the server.

We believe that this triangular model provides a lens through which we can examine prompts not just as one-off commands, but as **evolving artifacts of requirements expression, negotiation, and realization** which capture the main intentions of the developer [17]. As prompts are iteratively refined, attention may oscillate between specifying what is needed (Functionality and Quality), how it should be approached (General Solutions), and how exactly it should be executed (Specific Solutions). The developer thus incrementally reshapes the solution space. In this sense, even prompts focused on implementation remain grounded in a shifting understanding of *Functionality and Quality*.

Model Validation. Before devising an empirical research agenda around this model, we conducted an initial assessment of its applicability using a diverse set of prompt examples drawn from sources on the web and from the DevGPT dataset [37]. Selected examples are presented in Table 2. In addition, we analyzed a sample of 359 prompts extracted from DevGPT (~2% of the prompts in DevGPT) and examined the extent to which prompt content could be mapped to the dimensions of the Prompt Triangle.

To obtain the sample, we randomly extracted initial prompts from conversations contained in `20230803_095317_commit_sharings.json` and `20231012_233628_pr_sharings.json`. We focused exclusively on initial prompts, as the objective at this stage was not to investigate prompt evolution during interactions, but rather to assess whether the Prompt Triangle dimensions are observable in standalone prompt formulations. Prompt content was categorized using Claude Sonnet 4.6 and subsequently reviewed manually to verify consistency. All prompts and analysis artifacts are available in the replication package [5].

Our analysis shown in Table 1 demonstrates a strong tendency toward structured, multi-component prompts. Requirements (i.e., *Functionality and Quality*) appeared in 80.8% of prompts, establishing them as the foundational element of prompt construction. The predominant pattern ($n = 247$, 68.8%) incorporated all three components, indicating that users frequently provide comprehensive specifications of the solution. *General Solutions* (79.7%) are more commonly included than *Specific Solutions* (69.4%), suggesting that users prefer constraining the solution while preserving implementation flexibility. Co-occurrence analysis shows that requirements with at least one solution component accounted for all but two prompts. This asymmetry supports a way of thinking where specifying intent precedes the prescription of a solution. The secondary pattern of requirement-plus-general-solution prompts ($n = 39$, 10.9%) further supports a two-stage refinement process from “what” to “how” that may optionally extend to specific implementation details. Around 20% of prompts were not applicable as they did not relate to coding (e.g., “translate to Spanish”).

3.2 Prompting as a Tool for Requirements Validation and Verification

Requirements Engineering traditionally distinguishes between *validation* (ensuring we are building the right thing) and *verification* (ensuring we are building it right). In the context of AI-assisted chat-based development, structured prompts offer a novel mechanism to support both activities. By explicitly articulating *Functionality and Quality*, *General Solutions*, and *Specific Solutions*, prompts act as evolving artifacts of stakeholder intent and technical interpretation. Requirement *validation* occurs as the functionality component is refined in light of feedback and gradually reconciled with specific solutions, ultimately aligning the

■ **Table 1** Prompt Component Analysis (N=359).

Pattern	Component Present			n	%	% (excl. N/A)
	Req	Gen	Spec			
All Three	✓	✓	✓	247	68.8	85.2
Req + Gen	✓	✓		39	10.9	13.4
Req + Spec	✓		✓	2	0.6	0.7
Gen + Spec		✓	✓	0	0	0
Req Only	✓			2	0.6	0.7
Gen Only		✓		0	0	0
Spec Only			✓	0	0	0
Not Applicable				69	19.2	–
Total	290	286	249	359	100	100
(%)	(80.8)	(79.7)	(69.4)			

prompt with stakeholder needs. Meanwhile, requirement *verification* is observable in the iterative interactions and adjustments between the three prompt components, which mirror the ongoing process of checking that the system is being built correctly. Thus, prompts not only guide the AI in code generation, they serve as dynamic roadmaps for both validating what should be built and verifying how it is implemented.

This bears similarities to classical requirement validation and verification techniques. Atoum et al.’s recent systematic review [1] categorizes these into six approaches. Our method resembles prototyping, inspection, and testing-oriented techniques.

In terms of validation by *inspection*, the *Functionality and Quality* part of the Prompt Triangle serves a similar role to traditional requirements documents subjected to inspection. However, there are two distinctions: First of all, rather than discrete review meetings, the approach enables continuous, AI-mediated inspection through the iterative refinement between *Functionality and Quality* and *Specific Solutions*. The AI acts as an automated consistency analyzer that surfaces conflicts when it cannot reconcile high-level intent with implementation details, though the effectiveness of this detection depends on the AI’s interpretive capabilities and how explicit the formulation in the prompt is. Second of all, we expect the *Functionality and Quality* part of a prompt to be much more limited than a requirements document. An important part of inspections is the relationships between requirements [21]. We do not expect such relationships to become visible in individual prompts, but potentially as part of longer conversations.

Prototyping as a requirement validation technique was introduced by Boehm in 1984 [3]. In prototyping, working models of the software are created to validate that requirements meet stakeholder needs. One of the strengths of the Prompt Triangle is that it separates the solution components from the *Functionality and Quality* parts, thus enabling rapid iteration between *Functionality and Quality* and multiple solution formulations. Unlike traditional prototyping, which requires substantial development effort to explore alternative designs, the Prompt Triangle allows developers to experiment with different approaches in the *General Solutions* and *Specific Solutions* components while holding Functionality constant, generating executable prototypes within seconds rather than days [4]. When the AI fails to generate satisfactory code when a *Specific Solution* is provided, this failure signals either requirement ambiguity or an incorrect solution approach. Notably, this mechanism also works in reverse: when the AI successfully generates code that nonetheless fails to meet stakeholder needs upon review, this reveals gaps in the *Functionality and Quality* rather than implementation errors—a form of validation through executable prototyping.

■ **Table 2** Example prompts with classification of the information contained in them to the components of the prompt triangle. More examples can be found in the supplementary material [5].

Prompt	Functionality and Quality	General Solutions	Specific Solutions
Write a Python implementation of merge sort optimized for memory efficiency with time complexity analysis and error handling for edge cases including empty arrays. ¹⁾	Sort input values; Provide time complexity analysis; Handle error/edge cases; Optimized for memory efficiency	Use merge sort algorithm; Implement in Python	
Create a TypeScript function that validates email addresses with the following requirements: Must be RFC 5322 compliant Rejects disposable email domains Returns detailed error messages. Example of expected function signature and usage: <code>function validateEmail(email: string): { isValid: boolean; message: string }¹⁾</code>	Validate input; RFC 5322 compliance; Reject disposable email domains; Return detailed error messages	Use TypeScript	<code>validateEmail(email: string)</code>
I want to enable custom, unobtrusive update notifications and installs by customizing SPUUserDriver. Let's start by planning the custom UI we'll need. We'll ONLY work on UI. Create a plan for creating SwiftUI views that can show the various states that are required by SPUUserDriver. I think the best place for these to show up is in the macOS window titlebars on the top-right. Create a plan to put it there. Consult the oracle. ²⁾	Enable custom update notifications; Only UI work		Put UI in macOS window titlebars; Use SwiftUI views
Generate a Python function using pandas to: Read a CSV file. Drop null values. Convert a date column to datetime format. Group data by category and calculate average values. ³⁾	Read CSV file; Drop null values; Convert date column; Group data by category; Calculate average values	Use pandas library; Implement in Python	<code>Read with pandas.read_csv</code>
Write a FastAPI endpoint that: Accepts a POST request with JSON data (<code>{name: str, age: int}</code>). Validates the input. Returns a success message with the received data. ³⁾	Validate input; Return success message	Use FastAPI framework; Implement in Python	Define a POST endpoint; Accept JSON with <code>{name: str, age: int}</code>
Create a Python script using BeautifulSoup to: Fetch HTML from 'example.com/news'. Extract all headlines (h2 tags). Export results to a JSON file. ³⁾	Fetch HTML; Extract data from HTML; Export results to JSON	Use BeautifulSoup library; Implement in Python	Extract all <code><h2></code> tags

1) <https://margabagus.com/prompt-engineering-code-generation-practices/>

2) <https://mitchellh.com/writing/non-trivial-vibing>

3) <https://zencoder.ai/blog/vibe-coding-prompts>

In verification via *testing-oriented approaches*, prompts inherently produce testable artifacts. By maintaining explicit traceability between *Functionality and Quality* (what to test) and *Specific Solutions* (how it is implemented) within the same artifact, the Prompt Triangle enables test generation that simultaneously validates stakeholder intent and verifies implementation correctness. However, the quality of generated tests depends critically on whether the AI grasps the essential test scenarios implied by *Functionality and Quality* and the edge cases introduced by the *Specific Solutions*. This dual focus is difficult to achieve when requirements and code exist as separate artifacts. In many cases, developers instruct AIs to generate test cases for each new functionality automatically. Research on automated test case generation with LLMs is just emerging (see, e.g., [28]), but combining requirement and solution approaches in a prompt can help guide the generation towards more targeted tests of the technical solution.

3.3 Prompt Patterns and Prompt Iterations

Recent studies analyze prompt patterns, but none treat prompts as blends of requirements and solutions. DiCuffa et al. [11] identify seven patterns, such as *persona* (“you are”, “pretend to be”) or *recipe* (“step-by-step”, “guide”), focusing on structural conventions rather than semantic content. Siddiq et al. [30] investigate quality issues with developer prompts, finding they often suffer from ambiguity and insufficient context for producing high-quality output. While these anti-patterns are complementary to our findings, we believe the Prompt Triangle can alleviate some of the issues Siddiq et al. identified by making the prompt components explicit and providing a way to reason about their balance. Della Porta et al. [10] analyze how prompt patterns affect code quality across three dimensions: meta-prompting (zero-shot and few-shot), chain-of-thought, and personas. They find no statistically significant differences between combinations, but like others, do not examine prompt content systematically. Huang et al. [14] specifically address prompt engineering for requirements engineering, exploring techniques including multimodal prompting (e.g., UI mockups) and self-reflection prompts where LLMs provide confidence values. They propose using generated code to validate requirements, yet do not distinguish between requirement and solution elements within prompts themselves. Beyond individual prompts, Robino [26] proposes “conversation routines” to structure extended human-LLM dialogues through system prompts, applicable to software development contexts. Our prompt triangle provides the first systematic model decomposing prompts into requirement (Functionality and Quality) and solution (General, Specific) dimensions, enabling analysis of how these elements evolve through iterative refinement and therefore significantly goes beyond the state of the art in terms of analytical powers and the ability to craft high-quality prompts.

4 Hypotheses

The Prompt Triangle decomposes prompts into three complementary dimensions that characterize the information developers communicate when interacting with AI coding assistants: *Functionality and Quality*, *General Solutions*, and *Specific Solutions*. While this decomposition helps explain *what* information is present in prompts, understanding prompts as requirements artifacts also requires explaining *how* these components are used throughout the development process.

As discussed in the previous sections, prompts increasingly resemble evolving software artifacts that support activities traditionally associated with Requirements Engineering (RE). Following established RE practice [22], we distinguish between three fundamental activities:

validation, concerned with ensuring that the right system is being built; *verification*, concerned with ensuring that the system is built correctly; and *solution generation*, concerned with exploring and realizing implementation alternatives. While the Prompt Triangle captures the semantic structure of prompt content, these RE activities characterize the role such content plays during development interactions.

Reflecting on the relationships between Prompt Triangle components and RE activities raises several questions concerning how prompts evolve, how developers compose them, and how prompt composition may influence resulting software quality. Based on the conceptual arguments developed throughout this paper and prior observations of iterative AI-assisted development [2, 6], we formulate the following hypotheses:

- H1 (Prompt Evolution)** During iterative AI-assisted development, prompt content gradually shifts from emphasizing **Functionality and Quality** toward increasingly detailed **General Solutions** and **Specific Solutions** information.
- H2 (Developer-specific Prompt Composition)** The composition of Prompt Triangle components differs according to developer characteristics, such as programming experience and domain familiarity. In particular, developers with greater experience are expected to provide proportionally more **Specific Solutions** information.
- H3 (Prompt Components and RE Activities)** Prompt Triangle components are associated with different RE activities. Specifically, **Functionality and Quality** information is expected to occur more frequently during validation-oriented interactions, whereas **Specific Solutions** information is expected to occur more frequently during verification-oriented interactions.
- H4 (Prompt Alignment and Development Outcomes)** AI-assisted development interactions exhibiting stronger alignment between Prompt Triangle components and their corresponding RE activities are expected to produce higher software quality outcomes.

Together, these hypotheses define an empirical research agenda for studying prompts as evolving requirements artifacts. H1 and H2 focus on understanding how prompt composition emerges and changes across interactions and users, whereas H3 and H4 investigate whether these structures influence development processes and software outcomes.

5 Discussion

The power of chat-based interfaces is most obvious in vibe coding. Meske et al. [20] define vibe coding as the culmination of a historical process of intent mediation: “the level of abstraction required to implement solutions has evolved from hardware manipulation through conceptual modeling to prompting, shifting the mediation of developer intent from deterministic instruction to probabilistic inference.” To guide this inference, “goal-oriented intent expression” via prompts is necessary. Crucially, Meske et al. consider actual implementation details irrelevant. However, in the chat-based coding prompts we have studied, solution details as captured in the Prompt Triangle are almost always present.

This discrepancy suggests either that current practice remains anchored to solution-level control due to developer trust issues or LLM limitations, or that Meske et al.’s vision represents an idealized future state not yet achievable. Our Prompt Triangle embraces this reality by explicitly accommodating solution guidance alongside functional intent, rather than treating solution details as impurities to eliminate. As LLMs mature, we hypothesize the relative weight of Specific Solutions may decrease, but the tripartite structure will remain relevant as developers will always need some architectural control. Our work on H1 and H2 will identify how developers evolve prompts during their work, while H3 and H4 will reveal how requirements and solutions co-evolve and whether this strategy benefits code quality.

Taking this further, OpenAI’s Sean Grove proposes that prompts become executable specifications, arguing future programmers will write elaborate specifications that generative AIs turn into complex programs⁴. Counter-arguments exist: executable specifications have long existed [13, 12] but have arguably failed to achieve large-scale adoption, and making specifications fully executable constitutes programming itself. Moreover, unlike formal programming languages, natural language is ambiguous. Jackson considers this ambiguity a strength rather than weakness for natural language reasoning in AI [15]. LLMs arguably realize his vision by connecting natural language to vector space concepts. However, Jackson envisioned reasoning AIs, not code generators: the ability to “understand the world” via natural language does not directly translate to reproducibly creating programs fulfilling user needs. Much requirements engineering work addresses disambiguating unclear requirements [16].

Our work addresses this conflict directly: we study how requirements are captured in prompts with all their ambiguity and how engineers constrain generation by providing solution approaches. We believe prompt requirements must evolve through disambiguation and scaffolding. H1 and H3 will illuminate this process, while H4 will reveal the impact on code quality when validation and disambiguation happen early.

The tool landscape further validates our premise that prompts are not merely transient instructions but increasingly function as software engineering artifacts. For example, Kiro⁵ promises to turn “your prompt into clear requirements, system design, and discrete tasks.” Here, prompts come first and requirements follow, suggesting that the requirements are already embedded within the prompt alongside implementation guidance, architectural decisions, and domain knowledge. Kiro’s value proposition is therefore not creating requirements from scratch, but extracting, structuring, and disambiguating the information that already exists within the prompt. This pattern is not unique to Kiro. Modern AI-assisted development environments such as GitHub Copilot Chat, Claude Code, and OpenCode also preserve conversational context and support reusable prompt artifacts in the form of prompt files, skills, or similar templates. In these systems, prompts are often iteratively refined, versioned, and shared among team members, resembling traditional software artifacts such as design documents or user stories. For instance, developers frequently maintain prompt libraries that encode architectural constraints [19, 18] (e.g., “generate a REST API following hexagonal architecture with Spring Boot”), testing strategies (e.g., “write unit tests using Mockito with clear separation of concerns”), or organizational coding standards. This observation aligns with recent research that explicitly frames prompts as software engineering artifacts and argues that prompt engineering can be informed by requirements engineering principles [33]. In that view, a prompt is not just a query to a model; it is a specification vehicle that captures intent, constraints, quality expectations, and contextual assumptions.

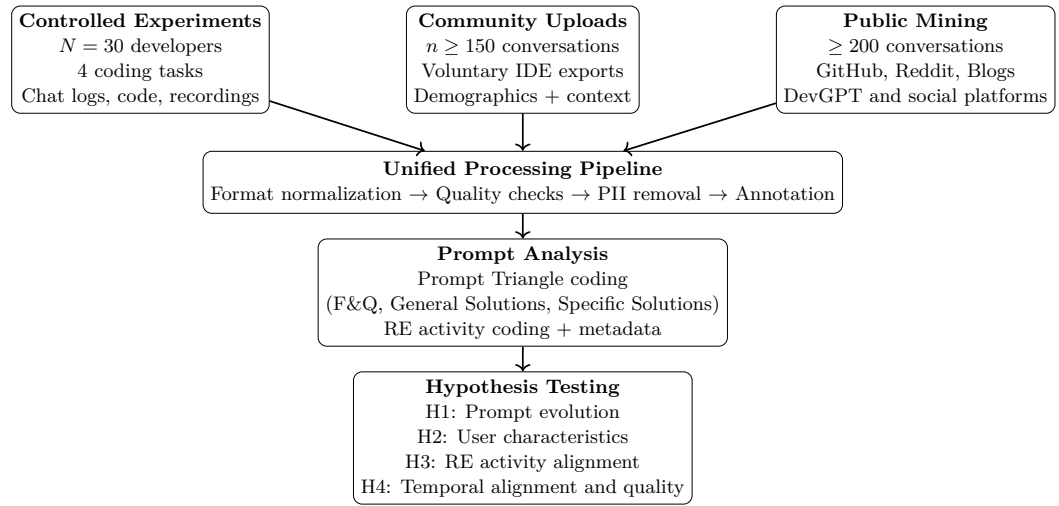
Another compelling argument lies in the role of prompts as boundary objects between stakeholders. In collaborative settings, prompts often encode both user intent and technical constraints, effectively bridging the gap between domain experts and developers [31]. For example, a product manager might articulate a feature through a structured prompt that includes user goals, edge cases, and acceptance criteria, which is then directly consumed by an AI system to generate implementation scaffolding. In this sense, prompts blur the distinction between requirements specification and implementation, collapsing phases of the traditional software lifecycle.

⁴ <https://www.youtube.com/watch?v=8rABwKRsec4>

⁵ <https://kiro.dev/>

■ **Table 3** Comparison of data collection methods.

Method	Data Type	Strengths	Limitations
Controlled Experiments	Structured, curated	Complete metadata, ground truth requirements, comparable tasks	Limited scale, potential observer effects
Community Uploads	Real-world, diverse	Authentic usage, varied contexts, scalable	Incomplete metadata, self-selection bias
Public Mining	Opportunistic	No recruitment overhead, naturally shared	Sparse, variable quality



■ **Figure 2** Overview of the mixed-method study design. Three complementary data sources (controlled experiments, community-sourced uploads, and publicly mined conversations) are integrated through a unified processing and annotation pipeline to investigate the proposed hypotheses.

These observations support a reconceptualization of prompts as structured, information-rich artifacts that encapsulate requirements, design intent, and domain knowledge.

6 Plans for Empirical Validation

We intend to employ a *multi-method data collection strategy* to construct a comprehensive corpus of developer–AI interactions. Our approach combines three complementary sources of data: (i) controlled experiments, (ii) community-sourced prompt exports, and (iii) mining of publicly available datasets. Figure 2 shows the complete vision for this study. It is described in more detail in a registered report [7].

Our triangulated design balances internal and ecological validity through three complementary data sources. Following Colnet et al. [9], who review methods for combining randomized trials and observational studies, our controlled experiment ($n = 30$) provides unconfounded causal estimates with full quality measures, while community uploads (target $n \geq 150$) and mined public datasets assess generalizability through observational data. Confirmatory hypothesis testing (H1–H4) is based on controlled experiment data; community uploads serve as co-primary replication data for H2 and supplementary validation for H1 and H3; mined datasets serve an explicitly exploratory role for pattern characterization only. Table 3 summarizes the collection approaches and their contributions to the corpus.

7 Conclusion

With chat-based development becoming more common, the role of software developers is starting to change. Instead of spending most of their time writing code line by line, developers are increasingly focused on thinking about how to communicate with AI systems. In many cases, writing a good prompt has become just as important as writing good code. These prompts are no longer simple instructions, they combine high-level requirements, design decisions, implementation hints, and constraints into a single artifact. Because of this, prompts deserve to be treated as first-class software engineering artifacts rather than temporary inputs to an AI tool. The Prompt Triangle provides a structured lens to characterize prompt content, enabling the analysis of prompt composition, evolution, and its relationship to requirements engineering activities and software outcomes. Based on this perspective, we formulate hypotheses on prompt evolution, developer strategies, and software quality. Looking ahead, we see this work as only the beginning. The ideas presented here raise several interesting questions about how developers refine prompts, how prompting strategies change throughout a development task, and how prompt evolution compares to the evolution of traditional software artifacts such as requirements and design documents. Future work will investigate these hypotheses to derive evidence-based practices and tool support for requirements-aware prompt engineering. Ultimately, we hope this will lead to practical guidelines, evidence-based prompting strategies, and tools that help developers create prompts that better capture software requirements and support high-quality AI-assisted development.

Data Availability Statement

The following materials are available in the supplementary material [5]: (1) a small set of prompts from sources on the web that were manually classified; (2) a set of 359 prompts randomly extracted from the DevGPT dataset and classified by Claude Sonnet 4.6; (3) a set of scripts to extract, classify, and visualize prompts from DevGPT.

References

- 1 Issa Atoum, Mahmoud Khalid Baklizi, Izzat Alsmadi, Ahmed Ali Otoom, Taha Alhersh, Jafar Ababneh, Jameel Almalki, and Saeed Masoud Alshahrani. Challenges of software requirements quality assurance and validation: A systematic literature review. *IEEE Access*, 9:137613–137634, 2021. doi:10.1109/ACCESS.2021.3117989.
- 2 Shraddha Barke, Michael B James, and Nadia Polikarpova. Grounded copilot: How programmers interact with code-generating models. *OOPSLA*, 7:85–111, 2023. doi:10.1145/3586030.
- 3 B.W. Boehm. Verifying and validating software requirements and design specifications. *IEEE Software*, 1(1):75–88, 1984. doi:10.1109/MS.1984.233702.
- 4 Markus Borg, Elizabeth Bjarnason, and Fabian Hedin. Vibe coding and the new prototyping playbook. *IEEE Software*, 42(6):12–16, 2025. doi:10.1109/MS.2025.3597032.
- 5 Shalini Chakraborty and Jan-Philipp Steghöfer. Dataset for “Prompts Blend Requirements and Solutions: From Intent to Implementation”, May 2026. doi:10.5281/zenodo.20329289.
- 6 Shalini Chakraborty and Jan-Philipp Steghöfer. Exploring prompts as mixed requirements and solutions artifacts. In *ICSE Companion*, 2026.
- 7 Shalini Chakraborty and Jan-Philipp Steghöfer. The prompt triangle: A registered report on prompts as hybrid artifacts, July 2026. doi:10.5281/zenodo.21351049.

- 8 Banghao Chen, Zhaofeng Zhang, Nicolas Langrené, and Shengxin Zhu. Unleashing the potential of prompt engineering in large language models: a comprehensive review. *arXiv preprint arXiv:2310.14735*, 2023. doi:10.48550/arXiv.2310.14735.
- 9 Bénédicte Colnet, Imke Mayer, Guanhua Chen, Awa Dieng, Ruohong Li, Gaël Varoquaux, Jean-Philippe Vert, Julie Josse, and Shu Yang. Causal inference methods for combining randomized trials and observational studies: a review. *Statistical Science*, 39(1):165–191, 2024.
- 10 Antonio Della Porta, Stefano Lambiase, and Fabio Palomba. Do prompt patterns affect code quality? a first empirical assessment of chatgpt-generated code. In *EASE*, pages 181–192, 2025. doi:10.1145/3756681.3756938.
- 11 Sophia DiCuffa, Amanda Zambrana, Priyanshi Yadav, Sashidhar Madiraju, Khushi Suman, and Eman Abdullah AlOmar. Exploring prompt patterns in ai-assisted code generation: Towards faster and more effective developer-ai collaboration. In *ICMI*, pages 1–7. IEEE, 2025.
- 12 Norbert E Fuchs. Specifications are (preferably) executable. *Software Engineering Journal*, 7(5):323–334, 1992. doi:10.1049/SEJ.1992.0033.
- 13 Ian James Hayes and Cliff B Jones. Specifications are not (necessarily) executable. *Software Engineering Journal*, 4(6):330–339, 1989.
- 14 Kaicheng Huang, Fanyu Wang, Yutan Huang, and Chetan Arora. Prompt engineering for requirements engineering: A literature review and roadmap. In *RE Workshops*, pages 548–557. IEEE, 2025. doi:10.1109/REW66121.2025.00081.
- 15 Philip Jackson. Understanding understanding and ambiguity in natural language. *Procedia Computer Science*, 169:209–225, 2020.
- 16 Ibrahim Khalil, Israr Ahmad, Uzair Rasheed, Wasi Haider Butt, and Zaeem Anwaar. Detecting cross-domain ambiguity in requirements through natural language processing, a systematic literature review. In *ICACS*. IEEE, 2025.
- 17 Jacob Krüger, Yi Li, Chenguang Zhu, Marsha Chechik, Thorsten Berger, and Julia Rubin. A vision on intentions in software engineering. In *FSE/ESEM, ESEC/FSE 2023*, pages 2117–2121, New York, NY, USA, 2023. Association for Computing Machinery. doi:10.1145/3611643.3613087.
- 18 Jenny T Liang, Melissa Lin, Nikitha Rao, and Brad A Myers. Prompts are programs too! understanding how developers build software containing prompts. *Proceedings of the ACM on Software Engineering*, 2(FSE):1591–1614, 2025. doi:10.1145/3729342.
- 19 Simerus Mahesh. Building scalable ai agents with modular prompt transpilation, July 2026. Google Developers Blog. Accessed: 2026-08-04. URL: <https://developers.googleblog.com/en/building-scalable-ai-agents-with-modular-prompt-transpilation/>.
- 20 Christian Meske, Tobias Hermanns, Esther Von der Weiden, Kai-Uwe Loser, and Thorsten Berger. Vibe coding as a reconfiguration of intent mediation in software development: Definition, implications, and research agenda. *IEEE Access*, 13:213242–213259, 2025. doi:10.1109/ACCESS.2025.3645466.
- 21 B. Nuseibeh, J. Kramer, and A. Finkelstein. A framework for expressing the relationships between multiple views in requirements specification. *TSE*, 20(10):760–773, 1994. doi:10.1109/32.328995.
- 22 Bashar Nuseibeh and Steve Easterbrook. Requirements engineering: a roadmap. In *Proceedings of the Conference on the Future of Software Engineering*, pages 35–46, 2000. doi:10.1145/336512.336523.
- 23 Partha Pratim Ray. A review on vibe coding: Fundamentals, state-of-the-art, challenges and future directions. *Authorea Preprints*, 2025.
- 24 Laria Reynolds and Kyle McDonell. Prompt programming for large language models: Beyond the few-shot paradigm. In *CHI*, pages 1–7, 2021. doi:10.1145/3411763.3451760.
- 25 Maxime Robeyns, Martin Szummer, and Laurence Aitchison. A self-improving coding agent. *arXiv preprint arXiv:2504.15228*, 2025. doi:10.48550/arXiv.2504.15228.
- 26 Giorgio Robino. Conversation routines: A prompt engineering framework for task-oriented dialog systems. *arXiv preprint arXiv:2501.11613*, 2025. doi:10.48550/arXiv.2501.11613.

- 27 Krishna Ronanki, Simon Arvidsson, and Johan Axell. Prompt engineering guidelines for using large language models in requirements engineering. *arXiv preprint arXiv:2507.03405*, 2025. doi:10.48550/arXiv.2507.03405.
- 28 Max Schäfer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. An empirical evaluation of using large language models for automated unit test generation. *TSE*, 50(1):85–105, 2023. doi:10.1109/TSE.2023.3334955.
- 29 Agnia Sergeyuk, Yaroslav Golubev, Timofey Bryksin, and Iftekhhar Ahmed. Using ai-based coding assistants in practice: State of affairs, perceptions, and ways forward. *IST*, 178:107610, 2025. doi:10.1016/J.INFSOF.2024.107610.
- 30 Mohammed Latif Siddiq, Simantika Dristi, Joy Saha, and Joanna CS Santos. The fault in our stars: Quality assessment of code generation benchmarks. In *SCAM*, pages 201–212. IEEE, 2024.
- 31 Philipp Steigerwald, Mara Stieler, Jennifer Burghardt, Eric Rudolph, and Jens Albrecht. Llars: Enabling domain expert & developer collaboration for llm prompting, generation and evaluation. *arXiv preprint arXiv:2605.10593*, 2026. doi:10.48550/arXiv.2605.10593.
- 32 Priyan Vaithilingam, Tianyi Zhang, and Elena L Glassman. Expectation vs. experience: Evaluating the usability of code generation tools powered by large language models. In *CHI*, pages 1–7, 2022. doi:10.1145/3491101.3519665.
- 33 Hugo Villamizar, Jannik Fischbach, Alexander Korn, Andreas Vogelsang, and Daniel Méndez. Prompts as software engineering artifacts: A research agenda and preliminary findings. In *PROFES*, volume 16361 of *LNCS*, pages 470–478. Springer, 2025. doi:10.1007/978-3-032-12089-2_32.
- 34 Andreas Vogelsang. Prompting the future: Integrating generative llms and requirements engineering. In *REFSQ Workshops*, 2024.
- 35 Huanting Wang, Jingzhi Gong, Huawei Zhang, and Zheng Wang. Ai agentic programming: A survey of techniques, challenges, and opportunities. *arXiv preprint arXiv:2508.11126*, 2025. doi:10.48550/arXiv.2508.11126.
- 36 Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837, 2022.
- 37 Tao Xiao, Christoph Treude, Hideaki Hata, and Kenichi Matsumoto. Devgpt: Studying developer-chatgpt conversations. In *MSR*, pages 227–230, 2024. doi:10.1145/3643991.3648400.