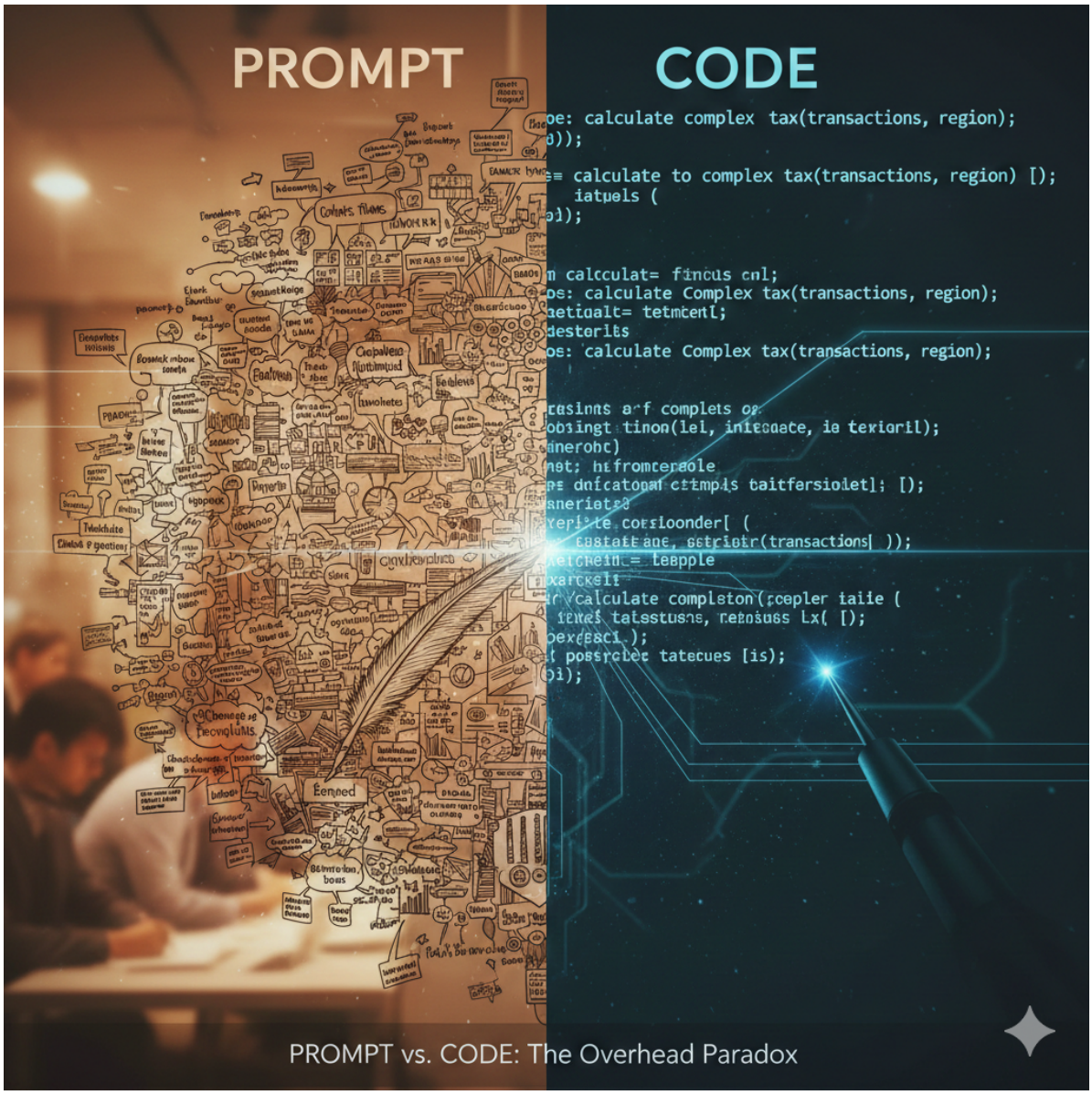
The Prompt-to-Code Paradox: Does Writing Human-Like Code Require a Prompt Longer Than the Code Itself?

Kabui, Charles

2026-02-13

Table of Contents



Introduction

A provocative question has emerged in the AI-assisted software engineering community: **To get an LLM to produce truly human-like code, do you need to write a prompt that is as long, or longer, than the code you would have written yourself?**

The short answer, backed by a growing body of research and practitioner experience, is: **not necessarily longer, but substantially more detailed than most developers expect.** The critical insight from the literature is that *structured specificity*, not raw length, is what separates prompts that produce fragile, average code from those that produce production-grade output. But achieving that specificity often results in prompts that rival or even exceed the length of the generated code, particularly for non-trivial tasks.

This article examines the research landscape as of early 2026, synthesizing findings from peer-reviewed studies, industry practitioners, and emerging frameworks to answer this question with rigour.

---

1. The Specification Gap: Why Natural Language Falls Short

Code is formal. Natural language is inherently ambiguous. Every prompt is an attempt to bridge this gap, what researchers call the **specification gap** between human intent and machine prediction.

1.1 Prompt Specificity Directly Impacts Code Quality

A 2023 study by Murr, Grainger, and Gao, *"Testing LLMs on Code Generation with Varying Levels of Prompt Specificity"* (arXiv:2311.07599), evaluated Bard, ChatGPT-3.5, GPT-4, and Claude-2 across 104 coding problems. Each problem was tested with four prompt types of varying specificity, including base descriptions, prompts with test cases, and prompts with explicit constraints.

Key findings:

- More specific prompts (those including test cases, constraints, and expected behaviours) consistently produced more accurate and efficient code.
- The performance gap between vague and specific prompts was significant across all tested models.
- GPT-4 with highly specific prompts approached near-human accuracy on many tasks, while the same model with vague prompts often produced incorrect or incomplete solutions.

This establishes a clear empirical relationship: **prompt detail is not a preference, it is a determinant of output quality.**

1.2 Chain-of-Thought Prompting Improves Code Logic

Liu et al. (2023), in *"Improving ChatGPT Prompt for Code Generation"* (arXiv:2305.08360), demonstrated that chain-of-thought (CoT) prompting with multi-step optimisations substantially improved ChatGPT's performance on both text-to-code and code-to-code generation tasks using the CodeXGlue benchmark.

The study found that **carefully designed, multi-step prompts guided the model through logical reasoning** before code generation, reducing errors in control flow, edge case handling, and API usage. The prompts required to achieve this, however, were significantly more elaborate than a simple task description.

1.3 Conversational Prompting Outperforms Automated Prompting

Shin et al. (2023/2025), in *"Prompt Engineering or Fine-Tuning: An Empirical Assessment of LLMs for Code"* (arXiv:2310.10508, accepted MSR'25), compared GPT-4 using three prompt engineering strategies (basic, in-context learning, and task-specific) against 17 fine-tuned models across code summarisation, generation, and translation.

A critical finding: GPT-4 with automated prompting was outperformed by fine-tuned models by 28.3 percentage points on the MBPP code generation dataset. However, a user study with 27 graduate students and 10 industry practitioners revealed that **conversational prompting, where humans iteratively provided explicit instructions, context, and corrections, significantly improved GPT-4's performance** beyond automated approaches.

This confirms that high-quality code generation is not about a single prompt. It requires an **interactive specification process** where the developer progressively narrows the model's solution space through detailed, human-in-the-loop guidance.

---

2. The Expertise Paradox: You Must Know the Code to Prompt for It

Research from human-computer interaction consistently reveals what Addy Osmani (Google, 2024) termed the **"Knowledge Paradox"**:

> AI tools help experienced developers more than beginners. Seniors use AI to accelerate what they already know how to do. Juniors try to use AI to learn what to do. The results differ dramatically.

This paradox extends directly to prompt writing:

- **To write a prompt detailed enough to produce correct, idiomatic, secure code, you must already understand how to write that code.** If you lack the knowledge, your prompt will be vague. If you have the knowledge, translating your mental model into structured English can feel as cognitively demanding as writing the code itself.
- Osmani's *"The 70% Problem"* (December 2024) documented that non-engineers could reach ~70% of a working solution with AI tools, but the remaining 30%: edge cases, error handling, security, performance, maintainability, required genuine engineering expertise that no amount of prompt crafting could replace.
- The paradox intensifies for complex tasks: describing 15 regional tax calculation rules in English takes more words than the 20 lines of code implementing them. The **business logic specification in natural language is inherently more verbose than its formal code equivalent.**

---

3. The "Curse of Instructions": Why More Isn't Always Better

While specificity improves output, **unstructured verbosity degrades it.** Research has confirmed a phenomenon dubbed the **"Curse of Instructions"** (documented by Anthropic and others):

- As the number of simultaneous instructions increases, model adherence to each individual instruction drops significantly.
- A study referenced in Anthropic's engineering guidance found that even GPT-4 and Claude struggle when asked to satisfy many requirements simultaneously. With 10+ detailed bullet points of rules, models reliably follow the first few and begin overlooking the rest.
- Long, unstructured prompts introduce **noise** that can distract the model, leading to higher error rates than shorter, focused prompts.

The research consensus is clear: **structured specificity, not length, is the variable that matters.** The goal is to be precise and organised, not merely verbose.

---

4. Emerging Solutions: Spec-Driven Development and Context Engineering

4.1 Spec-Driven Development (2025 - 2026)

The industry response to the prompt-quality challenge has coalesced around **spec-driven development**, treating AI prompts as formal software specifications rather than casual requests.

GitHub's analysis of over 2,500 agent configuration files (published January 2026) revealed that the most effective AI specifications consistently covered **six core areas:**

1. **Commands** — Full executable commands with flags, not just tool names
2. **Testing** — Frameworks, file locations, coverage expectations
3. **Project structure** — Explicit directory layout
4. **Code style** — One real code snippet beats three paragraphs of description
5. **Git workflow** — Branch naming, commit format, PR requirements
6. **Boundaries** — What the agent must never touch ("Never commit secrets" was the single most common helpful constraint)

Addy Osmani, in *"How to Write a Good Spec for AI Agents"* (January 2026), formalised a five-principle framework:

1. Start with a high-level vision and let the AI draft the details
2. Structure the spec like a professional PRD (Product Requirements Document)
3. Break tasks into modular prompts and context—not one monolithic prompt
4. Build in self-checks, constraints, and inject human expertise
5. Test, iterate, and evolve the spec continuously

The GitHub Spec Kit introduced a **four-phase gated workflow**: Specify → Plan → Tasks → Implement. Each phase requires validation before proceeding.

4.2 Context Engineering for AI Agents

Mohsenimofidi et al. (2025/2026), in *"Context Engineering for AI Agents in Open-Source Software"* (arXiv:2510.21413, accepted MSR 2026), studied the adoption of AI context files (such as `AGENTS.md`) across 466 open-source projects.

Key findings:

- There is no established content structure yet, significant variation exists in how context is provided (descriptive, prescriptive, prohibitive, explanatory, conditional).
- The information developers provide in these files is essentially **a specification document that guides AI behaviour**, confirming that the community is converging on the need for detailed, structured context.

- The field is actively studying which structural modifications positively affect generated code quality.

4.3 CNL-P: Controlled Natural Language for Prompts

Xing et al. (2025), in *"When Prompt Engineering Meets Software Engineering"* (arXiv:2508.06942), proposed CNL-P (Controlled Natural Language for Prompts), a system that applies software engineering principles to prompt writing:

- Precise grammar structures and strict semantic norms eliminate natural language ambiguity
- A **linting tool** checks prompts for syntactic and semantic accuracy, applying static analysis techniques to natural language for the first time
- Experiments showed that this structured approach enhanced LLM response quality through the synergy of prompt engineering and software engineering

This represents a formal recognition that **prompts are code**, they benefit from the same rigour, structure, and tooling we apply to software.

---

5. How Reasoning Models Change the Equation

Halim et al. (2025), in *"A Study on Thinking Patterns of Large Reasoning Models in Code Generation"* (arXiv:2509.13758), analysed how large reasoning models (LRMs) like OpenAI's o3, DeepSeek R1, and Qwen3 approach code generation.

Key findings:

- LRMs follow a **human-like coding workflow**, typically proceeding through problem understanding => planning => implementation => verification phases.
- More complex tasks elicit additional reasoning actions such as scaffolding, flaw detection, and style checks.
- Actions like **unit test creation and scaffold generation** strongly correlated with functional correctness.
- Lightweight prompting strategies that included **context-oriented and reasoning-oriented cues** improved code quality, suggesting that even with advanced reasoning models, structured prompts outperform vague ones.

This suggests an evolving landscape where newer models partially internalise the reasoning that older models required explicit instruction for—but structured context still helps.

---

6. The Prompt-to-Code Ratio: What the Evidence Actually Shows

Synthesising across the research, the relationship between prompt detail and code quality follows a clear pattern:

Task Complexity	Optimal Prompt Strategy	Typical Prompt:Code Ratio
Trivial (sort a list, format a string)	Short, direct instruction	~1:10
Moderate (CRUD endpoint, data transformation)	Specific with constraints, types, edge cases	~1:3 to 1:1
Complex (OAuth flow, business logic, architecture)	Full specification with examples, constraints, tests	~1:1 to 2:1
Production-grade (security, performance, maintainability)	Spec-driven with iterative refinement	Often >1:1

The pattern is clear: **as task complexity increases, the prompt-to-code ratio approaches and can exceed 1:1.** For production-grade code (the kind that handles edge cases, follows security best practices, adheres to style guides, and includes proper error handling) the specification effort often rivals writing the code manually.

However, the effort is not wasted. As Osmani observed:

> "If you spend 2 minutes writing the prompt, the code might require 10 minutes of fixing. If you spend 10 minutes designing the prompt, the code might require 1 minute of reviewing."

The total effort (prompt + review) is typically less than writing from scratch, even when the prompt itself is long. **The effort shifts from typing syntax to defining requirements**—a higher-leverage activity.

---

7. Where AI Definitively Saves Time (Regardless of Prompt Length)

The research is unambiguous that AI excels in specific scenarios regardless of prompt-to-code ratio:

1. **Boilerplate generation** — Short prompt ("Create a FastAPI CRUD app") produces hundreds of lines of repetitive but correct code.
2. **Breadth over depth** — The model draws on libraries, patterns, and APIs you haven't memorised.

3. **Refactoring** — "Make this more readable" (short prompt) => significant structural improvements.
4. **Test generation** — Given existing code, models generate comprehensive test suites efficiently.
5. **Documentation** — Models excel at explaining and documenting existing code.
6. **Exploration** — Rapid prototyping of multiple approaches to compare trade-offs.

In these scenarios, the prompt-to-code ratio is highly favourable (1:10 or better), and the AI provides clear time savings.

---

8. Practical Recommendations

Based on the accumulated research and practitioner evidence:

For Individual Developers

1. **Invest in specification, not just prompting.** Treat prompts as software specifications. Define inputs, outputs, constraints, error handling, and style before requesting code.
2. **Use structured formats.** Markdown with clear headings, bulleted constraints, and code examples outperforms prose paragraphs.
3. **Break complex tasks into focused sub-prompts.** Each prompt should target one function, one module, or one concern. Avoid monolithic prompts.
4. **Include examples.** One concrete input/output example is worth more than three paragraphs of description (few-shot prompting).
5. **Define what you don't want.** Explicit boundaries ("Do not use class components", "Never store passwords in plaintext") prevent common AI missteps.
6. **Iterate, don't one-shot.** Use conversational refinement. Review output, provide feedback, and regenerate.

For Teams and Organisations

1. **Adopt spec-driven workflows.** Use tools like GitHub Spec Kit's Specify => Plan => Tasks => Implement pipeline.
2. **Maintain `AGENTS.md` or `CLAUDE.md` files** with project-specific context, style guides, and boundaries.
3. **Invest in test infrastructure.** A robust test suite is a force multiplier—it gives AI agents a fast feedback loop for self-correction.
4. **Use the three-tier boundary system:** Always do / Ask first / Never do.

5. **Treat prompts as version-controlled artefacts.** Commit spec files alongside code.

Conclusion

The research confirms that generating human-like code from LLMs requires **substantially more specification effort than most developers initially expect.** This is not a flaw in the technology—it is a consequence of the fundamental gap between ambiguous natural language and precise formal logic.

However, the finding is nuanced:

- **It is not about raw prompt length.** Unstructured verbosity actively degrades output quality.
- **It is about structured specificity.** Clear constraints, examples, type information, and architectural guidance produce dramatically better code.
- **The effort is redistributed, not eliminated.** Time shifts from writing code to specifying requirements—a shift that, at scale, produces more maintainable and correct software.
- **The ratio varies by task complexity.** Simple tasks need minimal prompting. Complex, production-grade tasks often require specifications rivalling the code length.

The emerging discipline of **spec-driven development** and **context engineering** represents the software engineering community's recognition that prompt engineering is, at its core, **requirements engineering**—and like all requirements engineering, it demands rigour, structure, and domain expertise.

As Andrej Karpathy observed: *"English is becoming the hottest new programming language."* But as the research makes clear, writing effective "English programs" demands the same discipline we apply to any other programming language.

References

1. Murr, L., Grainger, M., & Gao, D. (2023). *Testing LLMs on Code Generation with Varying Levels of Prompt Specificity.*
2. Liu, C., Bao, X., Zhang, H., et al. (2023). *Improving ChatGPT Prompt for Code Generation.*

3. Shin, J., Tang, C., Mohati, T., et al. (2023/2025). *Prompt Engineering or Fine-Tuning: An Empirical Assessment of LLMs for Code.*

4. Denny, P., Kumar, V., & Giacaman, N. (2022). *Conversing with Copilot: Exploring Prompt Engineering for Solving CS1 Problems Using Natural Language.*

5. Halim, K., Teo, S.G., Feng, R., et al. (2025). *A Study on Thinking Patterns of Large Reasoning Models in Code Generation.*

6. Mohsenimofidi, S., Galster, M., Treude, C., & Baltes, S. (2025/2026). *Context Engineering for AI Agents in Open-Source Software.*

7. Xing, Z., Liu, Y., Cheng, Z., et al. (2025). *When Prompt Engineering Meets Software Engineering: CNL-P as Natural and Robust "APIs" for Human-AI Interaction.*

8. Zhao, J., Yang, D., Zhang, L., et al. (2024). *Enhancing Automated Program Repair with Solution Design.*

9. Osmani, A. (2024). *The 70% Problem: Hard Truths About AI-Assisted Coding.* Substack (Elevate).

10. Osmani, A. (2026). *How to Write a Good Spec for AI Agents.*

11. Osmani, A. (2026). *My LLM Coding Workflow Going Into 2026.*

12. GitHub Blog. (2023). *How to Write Better Prompts for GitHub Copilot.*

13. GitHub Blog. (2026). *How to Write a Great agents.md: Lessons from Over 2,500 Repositories.*

14. GitHub Blog. (2026). *Spec-Driven Development with AI: Get Started with a New Open-Source Toolkit.*

15. Willison, S. (2025). *Vibe Engineering.*
