Beyond Code-Centric

Agents Code but the Problem of Clear Specification Remains

Greg Ceccarelli

Co-Founder, SpecStory

June 15, 2025

Executive Summary

AI coding agents can now generate code on demand, moving the primary software bottleneck from development speed to specification clarity. Robert C. Martin's maxim ”specifying requirements so precisely that a machine can execute them is programming” resonates. Waterfall, Agile, and GitFlow all assume human interpretation. Each stumbles in today's multi-voice environment with agents.

Structured workflows (like Specflow) coupled with trunk-based development, unify intent, code, and tests, forcing teams to make domain knowledge explicit. AI excels at implementation but can't infer context or bridge contradictory specs. By orchestrating trunk-based flows, focusing on architecture and robust tests, and treating ”the specification itself” as first class alongside versioned code, teams can harness AI's speed without sacrificing clarity. Those who master collaborative specification in this era will own the future.

Table of contents

1 Part I – When Bottlenecks Shift

Software's bottleneck has moved: it's no longer about how fast we type but how clearly we think. Call it the abstraction shift. AI agents can churn out code on demand from high-level prompts[1], so progress now hinges on clarity of thought behind specification of what should be built.

Seventeen years ago in the first chapter of Clean Code[2], Robert C. Martin wrote: there will be code**.** He went on:

> "Indeed some have suggested that we are close to the end of code. That soon all code will be generated instead of written. That programmers simply won't be needed because business people will generate programs from specifications.
>
> Nonsense! We will never be rid of code, because code represents the details of the requirements. At some level those details cannot be ignored or abstracted; they have to be specified. And specifying requirements in such detail that a machine can execute them is programming. *Such a specification is code."*

This white paper revisits old problems, discusses what works, surfaces new pain points, and highlights how software practice must adapt.

For the abstraction shift to succeed it will require a reorganization of how we build software. The future points to trunk-based flow, with small, tight teams steering spec-driven agents**.**

In this model, humans focus on understanding user needs, making architectural tradeoffs, and precisely articulating intent. Implementation becomes the domain of agents. The most valuable people are the clearest thinkers: software composers[3] and conductors who will orchestrate agents to build systems once requiring much larger teams.

1.1 The Past Two Years

Few predicted LLMs would work so well, or that the killer use case would turn into the fastest growing SaaS company on record. Yet Cursor reportedly at ∼$500M[4] ARR as of May 2025 is exactly that.

The shift goes far beyond individual productivity gains[5]. Teams using LLMs report faster PR merges, quicker bug fixes, and bolder refactors–yet the 2024 DORA report shows[6] that wins are uneven, with∼40% of developers still wrestling with prompt churn, review drag, and trust gaps in AI-generated code.

So Natural language programming has partly arrived, but not in the way skeptics or optimists foresaw[7]. We thought natural language would become the programming language itself. Instead, it remains what it always was: the conversation that guides what gets programmed. What changed isn't the language of specification but who translates it into code. Martin's concept of "specifying requirements so precisely that a machine can execute them" has new meaning in practice.

The industry's experience unfolds in two parts. First, individual developers benefit from AI code generation (Cursor, Windsurf, Copilot, Claude Code, etc.), letting solopreneurs build functional software quickly[8] and often postponing costly early technical hires.

Second, *scaling* that software from "one to ten to one hundred" demands coordinated intent from many people to ensure viability, usability, profitability, and performance. Software remains a moving target as it evolves alongside users and markets.

As Diwank Singh reflected in his field notes[9] on shipping real code with Claude (which matches our own experience):

> "Think of traditional coding like sculpting marble. You start with a blank block and carefully chisel away, line by line, function by function. Every stroke is deliberate, every decision yours. It's satisfying but slow. Vibe-coding is more like conducting an orchestra. You're not playing every instrument—you're directing, shaping, guiding."

That works well for a single conductor. But when multiple conductors must coordinate the same orchestra, our old systems completely break down.

1.1.1 Why Today's Paradigms Break with Agents

Humans fill gaps that agents blast past. We infer context and ask clarifying questions. Agents require explicit guidance or they invent their own. This distinction sounds small but our entire ecosystem (tools, processes, methodologies) has always relied on *human* interpretation.

When a PM says, "Make it more user-friendly," or "Simplify the onboarding," a human developer books a call, asks questions, and iterates. Feed the same line to an AI agent and it invents solutions. No middle ground.

You can't engineer this away. It's baked into how agents execute prompts. The impact on teamwork is profound. In an environment where specifications must capture multiple stakeholders' perspectives (product vision, business realities, design constraints, user limitations, technical architecture) *any* ambiguity turns into a bug, an assumption, or a blocker.

2 Part II – Upgrading our approach for the Agent Era

Today's environment is code-centric. Tools like Jira, Linear and GitHub, methodologies like Agile, roles like PMs and Designers all funnel human-specified ideas into an engineering bottleneck. Engineers then translate these ideas into software. The shift to intent-centric development undercuts older models designed to manage human-limited bandwidth.

Clean Code does not dedicate a chapter to "specifications" because it declares where they belong. Traditional software development often treated specifications as static documents drafted before coding began. *Clean Code* instead suggests that code becomes clean because it is self-describing, with tests serving as executable verification.

This works when those writing the code deeply understand the business and user. But it assumes teams can hold the full system in mind and relies on human interpretation – something AI can't yet do[10], and which often fails when developers lack proper research or context.

At SpecStory, we see it firsthand: a quick prototype by one person may capture functional requirements yet miss deeper architectural and UX demands. The moment additional voices join, what worked seamlessly for one person becomes a coordination nightmare. Every assumption that was "obvious" to the original developer now requires explicit documentation and alignment

We often ask for *"one definitive file to copy/paste or @ reference in any chat UI to reset project memory for the next implementation session."* This indicates the deep fragmentation in our context. The solo flow that works for an individual and an agent fails under multi-voice input.

2.1 Waterfall's Rigidity

Waterfall seems perfect: specs, all upfront. In practice, book-length documents choke LLMs. They are lengthy, imprecise in AI terms, and oriented toward a final product rather than iterative building blocks. They also assume a single "authority" owns specifications.

AI-first teams quickly collide with questions waterfall can't handle: "Which model best handles pre-planning?" "How do we feed the codebase context to the AI?" If you hand a two-hundred-page specification to current generation AI agents, you will get a chaotic[11] implementation (if output is even possible).

Agents lack human context inference and Waterfall's static approach can't integrate multiple voices[12] effectively.

2.2 The Agile Paradox

Agile's super-power (lightweight autonomy) turns into its achilles-heel when AI joins the team. By favouring "working software over documentation," it keeps crucial design intent in hallway chat, stand-ups and tacit knowledge. That's fine for humans who can negotiate nuance on the fly, but agents can't attend retros or read between the lines.. As Purtell[13] notes:

> "Storing critical system information solely in human minds - and doing so more often as time goes on and AI becomes a bigger part of software - is not a good idea. The I/O bandwidth is low, the information degrades quickly, and collaboration scales poorly."

To harness agents without losing velocity, teams must pair Agile autonomy with rigorously captured, executable knowledge making specifications first-class citizens alongside code and conversation.

2.3 GitFlow Fragments Context at Scale

GitFlow[14] with its multiple long-lived branches epitomizes code-centric thinking. This made sense when engineering time was precious and changes were slower. Now, with agents generating code at high speed, each branch multiplies contexts the AI must reconcile.

A typical scenario: a feature touches `main`, `dev`, `feature-branch-1`, and `hotfix-branch-2`.

- When you prompt an AI with "implement feature X," is it a new branch? And which branch does it branch from?
- A human developer intuitively knows based on the conversation, the task at hand, and tacit knowledge.
- The AI agent sees four conflicting realities with no way to reconcile them.

Deciding which branch to use, and when to merge or rollback code becomes perpetual overhead.

2.4 Trunk Based Development Wins

At SpecStory we recognize three realities:

1. Agents are excellent at *implementation* given solid specifications.
2. Humans remain indispensable for design, prioritization, and judgment.
3. The gap between English language specification and code is narrowing.

We've adopted trunk-based development[15] as a preferred development methodology. Unlike traditional trunk-based teams, however, we rarely write code ourselves. Instead, each role contributes to a single shared repository:

- **PMs:** testable user behaviors (functional intent).
- **Designers:** coded constraints: spacing, components, interactions (design intent).
- **Architects:** explicit interfaces, contracts, and dependencies (technical intent).

By versioning specs, code, and tests together, we address the chaos of large interdependent features. "Where do I start?" becomes simpler because unfinished work cannot hide in branches. One branch enforces one truth. Near-instant implementation from an AI eliminates the need to defer integration. Cheaper to develop tests means broader and deeper test coverage is protecting the trunk from regressions. The requirement for explicit specifications ensures no knowledge remains hidden in people's heads.

Martin said, *"It is the responsibility of every software professional to understand the domain of the solutions they are programming."* Today, that responsibility intensifies. It's not enough to *know* the domain. You have to *articulate* it precisely so both humans and AI can evolve it. In trunk-based development, clarity is the ultimate constraint and there's no place to conceal ambiguity.

3 Part III – How Specflow Operationalises Trunk + Specs

After thousands of hours using agents with Cursor, Copilot, and Claude Code, we formalized **Specflow**[16], an open, flexible workflow that converts intent into software through structured planning and iterative execution. We use Specflow on top of trunk-based development.

Plan first, act second. Specflow accomplishes this in five steps:

1. **Pre-plan**: PMs define user outcomes; designers add patterns; engineers list constraints. A top reasoning model (Claude 4, GPT-o3) helps synthesize.

2. **Roadmap**: Everyone co-authors a markdown roadmap in the repo.

3. **Workplans**: Each roadmap phase is broken down into human and AI-executable tasks. This forces *implicit* knowledge out into the open.

4. **Execute**: Agents tackle tasks; anyone can intervene because context lives in git.

5. **Update & Refine:** Docs evolve with each cycle, capturing decisions and "as-built" truth in shared project memory.

Like Clean Code's emergent design, Specflow relies on code and tests as specifications, but agent-first development adds a crucial third layer: declared intent (roadmaps, workplans, docs, etc all in markdown) that both humans and AI can interpret.

This approach let three of us deliver a complete, end-to-end macOS alpha of our Studio product in just four weeks – an impossible timeline using traditional means.

3.1 Ad-Hoc = Adversity

Without this structure, projects spiral: vague prompts yield incorrect outputs, repeated revisions, regressions, difficult integrations, and unmanageable technical debt. "Vibe coding" by multiple roles in multiple places leads to fragmented results none of which align.

Diwank Singh at Julep describes this:

> "Without proper guardrails, you're not coding anymore—you're playing whack-a-mole with an overeager intern who memorized Stack Overflow but never shipped production code."

The core failure is context loss. Each AI prompt is relatively stateless, so a lack of shared structure means the system doesn't converge. And with multiple team members giving scattered instructions, fragmentation compounds.

3.2 Current Pain Points

Specflow imposes order but it also redistributes complexity. We've learned that while it addresses big issues, it demands mastery of micro-decisions that seasoned developers handle via intuition.

- **The context loading problem:** Before any prompt, we must decide *what* context to load. A senior developer instinctively knows to mention a custom state manager but omit a trivial logging utility. AI requires conscious decisions about every detail, to avoid either drowning it in noise or starving it of context.

- **Model choice micro-decisions:** We must also decide which model is best at each stage: GPT-o3 for architecture, Claude 4 for user stories, Gemini for implementation? Humans switch modes seamlessly: with AI, you have to pick models and prompts deliberately . A developer reading "implement user authentication" implicitly considers databases, APIs, forms, security, etc. With agents, each sub-decision must be explicit. Over-specify and you do needless work; under-specify and the AI guesses incorrectly.

- **Precision overhead:** LLMs understand "enterprise-ready" (SSO, RBAC, audit logs) better than most developers. But they don't know *when* to apply that knowledge. Humans leave scope unspoken, assuming teammates know whether they're building a disposable prototype or production grade software. LLMs don't ask and just assume, delivering battleships when you need dinghies or dinghies when you need battleships. The cost isn't typing out requirements. It's the *awareness* of every unspoken assumption that humans typically navigate through context clues and clarifying conversations.

- **Intervention and versioning trade-offs:** Deciding *when* to interrupt agents is tricky. Humans can sense small mistakes and pivot quickly. With agents, an early stop might waste potential progress whereas a late stop might produce a large tangle to unravel. Progress tracking also becomes more explicit. Developers often just "know" if a task is complete. For AI, you must define done criteria, commit points, branching, and documentation policies. Each agent turn might warrant commentary or a new commit. The overhead can be enormous if not carefully managed.

- **The maintenance burden:** Microsoft Research has shown that even the most advanced models achieve only 48.4% success on debugging tasks[17]. Agents generate code without global context creating "house of cards code" that appears complete but fails under real-world pressure.

3.3 The Insight

The future belongs to teams who can harmonize AI's precision requirements within human cognitive limits.

Until we create better intent-centric tools that reduce these micro-decisions, spec-driven development with agents remains powerful in theory but insufficient in practice.

Teams should spend energy shaping architecture, not typing file paths. Those who thrive distinguish between frictional micro-decisions and high-leverage macro-decisions. Yes, the overhead today is real but so are the gains when properly managed.

4 Part IV – Where the Human Edge (and ROI) Now Lives

In this new landscape, the leverage comes from humans doing what only humans can do. Yet the nature of that work has changed.

- **Domain modeling and architecture** still need human insight, especially now that implicit knowledge must be spelled out for AI. Humans see hidden connections and unstated needs that remain invisible to language models. But we have to convey these insights as specifications an agent can execute.

- **Intent specification and refinement** is now essential. Writing specs that are simultaneously precise enough for an AI to implement, yet flexible enough to allow solutions through iterative refinement, has become a specialized skill. This addresses the mentorship challenge: while 84% of developers learning to code use AI tools[18] Senior developers must now teach specification and architecture rather than syntax.

- **Quality judgment stays human, but it now works differently.** Rather than checking line-by-line code, we judge whether the output of our intent solves genuine user problems. CSET studies[19] show 48% of AI-generated code contains vulnerabilities, requiring 25-40% more review time[20]. But teams implementing proper review processes catch 75% of these issues[21].

- **Context preservation and evolution matter deeply**. The codebase must hold a coherent narrative arc that both humans *and* AI agents can traverse. In practice, that means curating institutional knowledge in executable form and elevating code quality standards so the code itself is optimised for AI readability: clean structure, rich metadata, and clear intent baked in from the start

4.1 The Sacred Rule: Never Let AI Specify Your Tests

One principle stands above others in AI-assisted development, echoing *Clean Code* fundamentals and the complexities of multi-voice teams. Singh emphatically states: *"Never. Let. AI. Write. Your. Tests."*. We think its *specification* that matters. If AI specifies them then control is ceded

Tests are much more than code checking code. They are executable specs capturing shared understanding of correct behavior.

Tests must reflect business rules, user needs, and critical insights from people who truly know the domain. Testing is also broader than running executable coded tests. Running software generated by agents must be evaluated by human users and is often done so continuously throughout its creation.

4.2 The New Stack: Intent + Code + Tests

The future stack we envision does not replace Clean Code principles. It evolves them for a world where implementation is cheaper with agents but coordination increasingly expensive.

- **Declared intent**: This is the first-class spec, reflecting all voices: product, design, architecture.

- **Code**: It's still important for clarity, maintainability and of course runtime behavior but now exists as just one possible execution of the clearly declared intent.

- **Tests**: They verify both the intention and its implementation, ensuring stakeholder alignment on the solution.

Instead of code being the primary specification as Clean Code advocated, it becomes one valid implementation of the intent specification. The code must still be clean, readable (by humans and agents), and maintainable but it's no longer the *only source of truth*.

4.3 The Craftsperson's Evolution

We are entering a new era of craftsmanship. Where *Clean Code* focused on individual and team excellence in writing code, AI-led development emphasizes collaborative specification. Tomorrow's craftsperson excels at:

- Architecting AI-implementable systems

- Writing specs that harmonize multiple stakeholder perspectives without diluting opinionated taste

- Maintaining consistent quality across human and AI contributions

- Orchestrating development at a higher abstraction level

Questions like "How many times did you prompt?" or "How do we keep the roadmap and final outputs aligned?" point to the shared shift: the bottleneck is no longer in writing code but in synthesizing intent for an AI to implement, and validating that the intent has been faithfully realized by the AI.

Teams that adapt to this new reality and are able to create a "team-wide flow state" rather than forcing everyone into ad-hoc prompting will flourish. Trunk-based development anchored by human specifications and executed by AI agents addresses context fragmentation and ensures quick, high-fidelity feedback. It forces alignment across multiple voices before and after code is generated.

Industry economics support this transition: IBM's 2024 study[22] found 47% of companies already seeing positive ROI from AI investments, with annual returns of 2,400 – 3,600% for properly implemented systems[23].

We do *not* foresee a no-code or low-code future. Rather, we see high-leverage code, where human intelligence from multiple domains directs AI to build things neither could achieve alone. The *specification* is still code, just a different kind: written by teams for a new interpreter that can be made to understand collective intent, bridging the gap between human creativity and machine efficiency.

Robert Martin was right: there will always be code. In fact, the specification itself is code. In the AI era, the spec must capture not just one visionary's plan but the combined intent of everyone necessary to deliver genuine value.

Teams that master the art of iterative specification to orchestrate agents will own the future.

5 Acknowledgments

The author thanks Dr. Cat Hicks, Jake Levirne, Sean Johnson and Akshay Bhushan for their thoughtful review and feedback on this white paper.

6 Endnotes

[1] Hou, W., & Ji, Z. (2024). **A systematic evaluation of large language models for generating programming code** (Version 1) [Preprint, *arXiv*]. arXiv. https://arxiv.org/abs/2403.00894

[2] Martin, R. C. (2008). *Clean code: A handbook of agile software craftsmanship*. Pearson.

[3] Bhushan, A., Ceccarelli, G., & Levirne, J. (2024, November 13). *The rise of the software composer: A new era of software creation*. Tola Capital. https://tolacapital.com/2024/11/13/the-rise-of-the-software-composer-a-new-era-of-software-creation

[4] Hammond, G. (2025, May 5). *Maker of AI "vibe coding" app Cursor hits $9 billion valuation*. *Financial Times*. https://www.ft.com/content/a7b34d53-a844-4e69-a55c-b9dee9a97dd2

[5] Vasudevan, V. (2025, April 29). *Github Copilot adoption trends: Insights from real data*. Opsera Blog. https://www.opsera.io/blog/github-copilot-adoption-trends-insights-from-real-data

[6] DeBellis, D., Storer, K. M., Villalba, D., Irvine, M., & Castillo, K. (2024). *2024 Accelerate State of DevOps Report* (Version 2024.2). Google Cloud. https://dora.dev/research/2024/

[7] Stack Overflow, *2024 Developer Survey*, section "AI in the development workflow," chart "Which parts of your development workflow ...," accessed June 8 2025, https://survey.stackoverflow.co/2024/ai (showing that **82 percent** of developers who already use AI tools rely on them specifically to **write code**); Wenpin Hou and Zhicheng Ji, "A Systematic Evaluation of Large Language Models for Generating Programming Code," *arXiv* preprint arXiv:2403.00894 v1 (cs.SE), March 1 2024, Figure 2, https://arxiv.org/abs/2403.00894v1 (reporting that GPT-4's five-attempt success rate **jumps from 15 percent under a basic "repeated prompt" strategy to 86 percent under an optimized "feedback CI" strategy**, underscoring how strongly outcomes hinge on prompt design).

[8] Code Coup. (2025, May 6). *How I built 18 MVPs in 6 months using Cursor AI* [Blog post]. Coding Nexus. https://medium.com/coding-nexus/how-i-built-18-mvps-in-6-months-using-cursor-ai-d3021252824a

[9] Tomer, D. S. (2025, June 7). *Field notes from shipping real code with Claude*. **diwank's space**. https://diwank.space/field-notes-from-shipping-real-code-with-claude

[10] Yiu, E., Kosoy, E., & Gopnik, A. (2023). Transmission versus truth, imitation versus innovation: What children can do that large language and language□and□vision models cannot (yet). Perspectives on Psychological Science, 19(5), 874–883. https://doi.org/10.1177/17456916231201401

[11] Lu, Y., Sachan, D. S., Li, Z., & Zettlemoyer, L. (2023). *Lost in the middle: How language models use long contexts* (arXiv:2307.03172). arXiv. https://arxiv.org/abs/2307.03172 — "We find that performance can degrade significantly when changing the position of relevant information, indicating that current language models do not robustly make use of information in long input contexts. In particular, performance is often highest when relevant information occurs at the beginning or end of the input context, and significantly degrades when models must access relevant information in the middle of long contexts."

[12] Jin, H., Huang, L., Cai, H., Yan, J., & Chen, H. (2024). *From LLMs to LLM-based agents for software engineering: A survey of current, challenges and future* (arXiv:2408.02479). arXiv. https://arxiv.org/abs/2408.02479 — “**It is still in its early stage for a unified standard and benchmarking to qualify an LLM solution as an LLM-based agent in its domain,**

[13] Purtell, J. (2025, March 18). *Specification engineering.* **Prolegomena**. https://www.joshuapurtell.com/posts/spec_eng/

[14] Driessen, V. (2010, January 5). *A successful Git branching model.* nvie. https://nvie.com/posts/a-successful-git-branching-model

[15] Hammant, P. (n.d.). *Trunk based development: Introduction.* TrunkBasedDevelopment.com. Retrieved June 8, 2025, from https://trunkbaseddevelopment.com/

[16] SpecStory. (n.d.). *Specflow: Structure for building with AI agents*. Retrieved June 8, 2025, from https://www.specflow.com

[17] Lardinois, F. (2025, April 10). *AI models still struggle to debug software, Microsoft study shows*. TechCrunch. https://techcrunch.com/2025/04/10/ai-models-still-struggle-to-debug-software-microsoft-study-shows/

[18] Stack Overflow. (2024). *2024 Stack Overflow developer survey: AI*. https://survey.stackoverflow.co/2024/ai

[19] Center for Security and Emerging Technology. (2024). *Cybersecurity risks of AI-generated code*. Georgetown University. https://cset.georgetown.edu/publication/cybersecurity-risks-of-ai-generated-code/

[20] DevOps.com. (2024). *AI in software development: Productivity at the cost of code quality?* https://devops.com/ai-in-software-development-productivity-at-the-cost-of-code-quality/

[21] Dark Reading. (2025). *Will AI code generators overcome insecurities in 2025?* https://www.darkreading.com/application-security/will-ai-code-generators-overcome-their-insecurities-2025

[22] IBM. (2024, December 19). *IBM study: More companies turning to open-source AI tools to unlock ROI* [Press release].

[23] Donahoe, T. (2024, September). *Research: Quantifying GitHub Copilot’s impact in the enterprise with Accenture*. The GitHub Blog. https://github.blog/news-insights/research/research-quantifying-github-copilots-impact-in-the-enterprise-with-accenture/