Dear Researchers: The craft that I have practiced for the last twenty years no longer exists.

What happens when AI agents become operational actors in software engineering?

Uwe van Heesch, TH Köln, uwe.van_heesch@th-koeln.de

Preprint version

This is the author's preprint of an article forthcoming in the *Dear Researchers* column of the *Journal of Systems and Software*. The final published version is available at: https://doi.org/10.1016/j.jss.2026.113000

A few weeks ago, I received a message from a close friend who is also the best developer I know, a lead engineer with more than twenty years of experience. He wrote: "The craft that I have practiced for the last twenty years no longer exists." He was referring to programming. He described how he had implemented a seemingly trivial feature, a data export in a web UI, something he would normally have estimated at thirty minutes of work. Instead, he typed a single prompt into Kiro, Amazon's coding agent that uses Anthropic's frontier models. Thirty seconds later, the feature was done, including UI, translations, and categorization. The result was correct. There was almost nothing to fix. What unsettled him was not that the tool had failed. It was that it had worked. At first glance, this may sound like the end of human software engineering. It is not. But something fundamental is changing, and I believe we need to talk about it, honestly and with a sense of urgency.

My central claim is this: the core of software engineering is shifting from implementation to intent, from writing code to defining constraints, encoding quality, and validating outcomes. Others have observed a similar shift from the perspectives of spec-driven development, AI-supported toolchains, and the growing need for formal or semi-formal specifications[1][2][3]. My perspective is complementary: I examine this transformation from everyday software engineering practice and from the operational integration of agents into real development workflows.

What follows is not another celebration of AI productivity. It is a practitioner's account of what breaks when autonomous agents enter real development teams. It closes with three open questions about process design, expertise shift, and engineering education that I believe require serious empirical research. I am fully aware that the use of LLMs, coding agents, and semi-autonomous systems raises serious concerns around security and data privacy, and environmental sustainability. These concerns are legitimate and must be addressed

[1] B. Böckeler, "Understanding Spec-Driven-Development: Kiro, spec-kit, and Tessl" 2025. Available online.

[2] GitHub, "Spec-driven development with AI," 2025. Available online.

[3] B. Congdon, "The Coming Need for Formal Specification" 2025. Available online.

rigorously. This article, however, focuses on a different dimension of the transformation: the changing nature of the work itself.

Over the past four years, I have worked as a software architect and DevOps engineer in industry. The company develops software for corporate learning and education, has roughly 90 employees, 15 of whom form the software development team. I have designed systems, built CI/CD pipelines, debugged production incidents, and spent countless hours writing and reviewing code. The company was an early adopter of AI-supported coding, which allowed me to follow the evolution of these tools closely and in a realistic production setting. It started with copy-pasting code to and from ChatGPT, moved to IDE-integrated assistants, then to commercial AI coding assistants and AI-powered development environments such as GitLab Duo, Amazon Q Developer, and Cursor, followed by custom integrations via LiteLLM proxies and AWS Bedrock, and eventually led me to agentic systems such as Claude Code. Each step increased the level of automation, and each step changed how I thought about my own role. At some point, the question was no longer whether the tool could help me write code. The question became how much of my work I could automate or delegate, and what would remain that was uniquely mine.

With modern coding agents, much of my daily work has changed. Some tasks I can fully delegate, others require iteration, but in all cases, the agent carries a significant share of the effort: implementing features, writing and executing tests, analyzing CI/CD failures, investigating performance issues, updating documentation, reviewing pull requests, and applying architectural rules across components. What surprised me most was not the code generation itself, but what happened when I connected the agent to the tools I use every day via MCP (Model Context Protocol)[4]. Suddenly, the agent could operate across the entire system, reading and updating Jira tickets, analyzing GitLab pipelines and logs, interacting with Kubernetes clusters, accessing cloud environments, and querying monitoring systems like Prometheus or Grafana. At that point, the agent was no longer just a coding assistant. It had become an operational actor, and I found myself shifting from writing code to defining the boundaries within which the agent was allowed to operate.

But capability alone did not produce quality. The agent's early failures were revealing. It did not struggle with programming itself. It struggled with conventions that had never been written down. Early on, much of the output was wrong in ways that only someone familiar with the project would notice: tests for RESTful HTTP controllers that ignored our established patterns, input validation that diverged from internal conventions, SQL queries that bypassed the company's predefined views and functions. Even with today's stronger models, the output without explicit guidance remains generic: functional, but not how an experienced team member would do it.

[4] Anthropic, "Model Context Protocol" 2024. Available online.

This taught me something important. Effective delegation requires that the agent can validate its own work against explicit criteria. Everything I previously applied implicitly, what "done" means in terms of functional completeness and test coverage, what "acceptable quality" looks like in terms of static analysis thresholds and architectural conformance, which invariants must hold across every change – all of this must be made explicit.

In practice, this meant that I had to translate my tacit knowledge into agent configuration, reusable skills, concrete examples, and automated validation mechanisms. I gave the agent explicit references to existing implementations, encoded the relevant conventions in local agent skills and configuration, and added conformance tests to check whether these conventions were actually followed. I also relied on integration tests to detect regressions, static analysis rules, screenshot-based UI validation, architectural guardrails, architectural decision records, and API contract checks. The CI/CD pipeline became the central authority that determines whether the agent's output meets the defined acceptance criteria.

The quality of the agent's output, I realized, is directly proportional to the quality of the constraints I define. The agent still wrote much of the code, but the definition of an acceptable solution no longer depended on my implicit judgment alone, but on explicit guidance, concrete examples, and executable validation. The software development craft, in my experience, will not disappear. It will move from writing code to encoding intent.

To explore the limits of this approach, I conducted an experiment. I used OpenClaw[5] to create a fully autonomous agent, one that, unlike tools where I directly instruct the agent, operates with its own lifecycle. It continuously checks for new tasks, processes them independently, communicates via chat interfaces, and maintains its own execution loop. I configured the agent to behave like a developer in a Scrum team. The setup was intentionally realistic: the agent ran locally in a Docker environment, had its own GitHub account, used separate credentials for all systems, processed backlog items, created pull requests for review and reviewed pull requests from other developers. My initial idea was simple: the agent works like a team member, and humans review and approve its work.

What happened next was not surprising in principle but seeing it in action was striking. The tickets on the Scrum board moved from left to right within minutes. The agent worked continuously, day and night, processing tasks significantly faster than any human could. Pull requests accumulated rapidly. The pattern that emerged was unambiguous: the human reviewer became the bottleneck. The system did not slow down because of technical complexity. It slowed down because of human intervention points. Not only could I not keep up with reviewing the agent's output, but the sprint backlog could not be filled fast enough by human planners to keep the agent occupied. I began to question whether the process I had designed around it was fundamentally flawed. What I observed was, in essence, a throughput problem in a sociotechnical system. The automated component had no practical capacity

[5] OpenClaw, "Your own personal AI assistant" 2025. Available online.

constraint, while the human component did. The review step, originally designed as a quality gate, had become a flow constraint.

This observation has implications well beyond my experiment. Process models such as Scrum are designed for human communication, human coordination, and human cognitive limits. They are highly effective precisely because they optimize for these constraints. But once agents perform a significant portion of the work, these assumptions break down. Agents do not require synchronization meetings, do not experience cognitive overload, do not need breaks, and can process tasks in parallel. When such agents operate within human-centric processes, systemic friction emerges. In my experiment, tasks moved across the board without human interaction. The process itself became the limiting factor. The conclusion is straightforward: we are applying process models optimized for human teams to systems that are no longer human-dominated. This is not a tooling problem. It is a process design problem.

The same mismatch appears at the organizational level. My friend, in his original message, pointed out that his AI-augmented productivity now exceeds what his organization can absorb. The surrounding structures, the roles, responsibilities, and decision processes, were not designed for this level of throughput. Work is completed faster than it can be validated, decisions are delayed because coordination takes time, and available capacity cannot be translated into delivered value. I see this in my own work as well. The technical capability scales rapidly, but the organizational structures around me do not. Teams that fully embrace AI-based development will inevitably start to remove unnecessary coordination overhead, redesign their processes, and optimize for throughput and validation rather than communication. In such environments, agents perform most operational tasks, humans define constraints and evaluate outcomes, and quality is enforced through automated mechanisms. This is not a distant scenario. The underlying capabilities already exist.

The transformation I have described is not hypothetical. It is observable in practice today. However, we lack a systematic understanding of its implications. I was invited to write this column as a means of communication between practitioners and researchers. So, dear researchers, here are the questions I believe matter most:

First, what do software development process models look like when autonomous agents perform the majority of implementation work? Specifically, how should validation, review, and approval workflows be restructured so that human oversight remains meaningful without becoming a throughput constraint? Existing models were designed for human teams, and they may not scale to hybrid or agent-dominated environments.

Second, where does human expertise shift as implementation becomes automated? Which cognitive tasks, such as requirements analysis, architectural reasoning and decision making, constraint definition, and system governance, become more critical, and how do we develop and measure proficiency in them? This shift needs to be studied empirically, not assumed.

Third, and probably most pressing: what should we teach the next generation of software engineers? Experienced engineers benefit enormously from AI because they possess the domain knowledge and judgment to direct these tools effectively. But if the path to that expertise has traditionally run through years of manual implementation, how do we build equivalent competence in a world where that path is disappearing? Should curricula emphasize problem framing, system design, quality definition, and the configuration of autonomous systems over manual coding and framework-specific knowledge? Educational programs are being designed now. We cannot afford to delay this discussion.

The craft of programming will not disappear. But it will no longer be where it used to be. If we continue to look for it in code, we will conclude that it has disappeared. If we look at how systems are defined, constrained, and validated, we will find that it has simply moved. The question is not whether this shift will happen. The question is whether we are prepared to understand and shape it.

Special thanks to Olaf Zimmermann and Austin Z. Henley for providing multiple rounds of feedback to improve this article.