

When Review Alone No Longer Scales: Layered Supervision in AI-Assisted Software Engineering

Markus Stolze ✉ 

OST Eastern Switzerland University of Applied Sciences, Rapperswil, Switzerland

Mirco Strässle ✉

smartive AG, St. Gallen, Switzerland

Abstract

AI-assisted development tools enable software engineers to generate implementations at substantially higher speed and volume than in traditional workflows. Software teams have long relied on guardrails – standing control mechanisms such as code review, linting, testing, and CI/CD pipelines – to maintain quality and coordination. High-throughput AI-assisted generation increases pressure on these guardrails – straining their capacity to keep pace with the volume and rate of generated changes – and reshapes how organizations supervise development workflows, yet relatively little is known about how existing guardrails evolve in response.

We conducted a qualitative interview study with five software engineering practitioners, situated within a broader practitioner survey. Our findings indicate that organizations distribute the work of supervision across multiple guardrail layers: preventive guardrails (produced by externalizing architectural intent and conventions into machine-interpretable form), executable guardrails (linting, testing, and CI/CD repurposed as scalable supervision infrastructure), and human oversight (shifting from line-by-line inspection toward supervisory interpretation focused on architectural reasoning, explainability, and long-term maintainability). We characterize this as a transition from review-centric guardrails toward *layered supervision*, in which no single guardrail carries the supervision load alone.

2012 ACM Subject Classification Software and its engineering → Software development techniques; Software and its engineering → Software verification and validation; Human-centered computing → Empirical studies in HCI

Keywords and phrases AI-assisted software development, AI coding tools, validation, code review, software teams, guardrails

Digital Object Identifier 10.4230/LIPIcs.ESEM.2026.90

Category Software Engineering in Practice Track Paper

Supplementary Material *Dataset (Evidence table, codebook, and survey instrument):*

<https://doi.org/10.5281/zenodo.21611622>

Acknowledgements We thank the interview participants for their time and candor, and colleagues at OST who provided feedback on early drafts and the survey instrument.

1 Introduction

AI-assisted development tools enable software engineers to generate implementations at substantially higher speed and volume than in traditional development workflows. By *AI-assisted development tools* we refer to large language model-based coding assistants and agentic development systems – for example, IDE-integrated assistants, chat-based code generation tools, and autonomous coding agents – that generate, modify, or refactor production code based on natural-language or code-context input. Recent advances in such tools have intensified discussions regarding productivity gains, code generation quality, and changing



© Markus Stolze and Mirco Strässle;
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. 90; pp. 90:1–90:12



Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

software engineering practices [14, 19]. At the same time, AI-assisted generation also raises new challenges regarding how generated artifacts are constrained, validated, and supervised within organizational development environments.

Software engineering workflows have long relied on *guardrails* – standing control mechanisms such as code review, linting, testing, CI/CD pipelines, and architectural validation – to support software quality, coordination, and maintainability [10, 16, 7]. AI-assisted generation, however, substantially increases implementation throughput while producing artifacts that may remain difficult to validate contextually despite appearing locally plausible or functionally correct [17]. Organizations therefore face growing pressure to supervise AI-assisted development at scales that cannot rely on downstream human review alone. Throughout, we use *guardrail* for any standing control that constrains or checks generation, whether preventive (e.g., specifications, steering files) or detective (e.g., linting, tests, code review), and whether enacted by automated tooling or by humans, and *supervision* for the overarching function of monitoring and steering generation that these guardrails collectively serve. We adopt this term deliberately over narrower alternatives such as *quality assurance mechanism* or *software testing practice*, which do not naturally extend to preventive, upstream constraints such as specifications or steering files that act before any output exists to test.

While prior research has focused strongly on productivity, developer interaction with coding assistants, and validation of generated code [14, 19, 17], comparatively few studies examine how organizations operationally constrain and supervise AI-assisted software engineering workflows. In particular, relatively little is known about how existing software engineering guardrails evolve in response to high-throughput AI-assisted implementation activity.

To address this gap, we conducted a qualitative interview study with five software engineering practitioners.

2 Related Work

2.1 AI-Assisted Software Engineering

Early empirical work on AI-assisted coding focused on productivity and developer–tool interaction, including controlled studies of GitHub Copilot [14], qualitative studies of developer validation behavior [17], and broader practitioner studies [19]. More recent industry-scale evidence indicates that AI-assisted development now produces code at a volume incompatible with exhaustive human inspection: the 2025 DORA report finds approximately 90% developer adoption and frames AI as a multiplier of existing engineering capability rather than a standalone productivity intervention [5], while a large-scale study of 302,600 AI-attributed commits across 6,299 GitHub repositories reports that AI-introduced quality issues persist and accumulate as technical debt [12]. Studies of agentic coding tools document mixed effects and call explicitly for provenance, supervision, and selective deployment [1], while complementary qualitative work surfaces practitioner concerns about trust and governance in agentic workflows [18]. These quality-risk findings are not unanimous, however: a preregistered two-phase experiment with 151 professional developers found no significant difference in subsequent maintainability – measured by independent developers evolving the resulting code – between AI-assisted and unassisted contributions [3], suggesting that downstream quality effects depend on task and context rather than being a uniform property of AI assistance. Our study characterizes how organizations operationally respond to these pressures.

2.2 Externalized Constraints and Executable Governance

Software engineering has long relied on linting, testing, CI/CD pipelines, and architectural validation to support quality and coordination [10, 16]. A complementary thread examines the externalization of organizational and architectural intent into machine-interpretable form. Foale et al. provide the first large-scale empirical study of Policy-as-Code adoption across 399 open-source repositories [7], and practitioner-oriented work on architectural fitness functions and architecture-as-code similarly emphasizes a shift from governance-by-inspection toward governance-by-rule [8]. In AI-assisted development specifically, recent work examines developer-provided context artifacts such as AGENTS.md files, Cursor rules, and structured specification files as configuration substrates for agentic tools [9] – the same authors have since retitled a later revision of this study *Harness Engineering for Agentic AI Coding Tools* (arXiv:2602.14690), converging with practitioner discourse that treats a coding agent’s harness, everything beyond the model itself, as an attempt to “externalise and make explicit what human developer experience brings to the table” [2], a formulation that closely echoes our own use of externalization for the preventive and executable guardrail layers below. Osmani situates this harness one layer below what he terms loop engineering, the design of automated, self-triggering agent cycles operating with reduced moment-to-moment human involvement [13]; our third layer, human oversight, addresses precisely the coordination question this raises as agentic workflows move toward such loops. Our study examines how such externalized constraints proactively guide AI-assisted generation upstream of review.

2.3 Human Oversight of AI-Assisted Development

A growing literature examines developer trust, automation bias, and oversight behavior in AI-assisted workflows. Qualitative studies report that developers often over-trust AI suggestions and may reduce vigilance when code appears plausible, particularly with respect to security [11]. Broader work on human oversight of AI systems, including Article 14 of the EU AI Act, distinguishes monitoring, interpretation, and intervention as separable oversight functions [6]. Our findings contribute to this discussion by characterizing how oversight in AI-assisted software engineering shifts from post-hoc inspection toward supervisory interpretation focused on architectural reasoning, contextual appropriateness, and long-term maintainability.

3 Methodology

3.1 Study Design and Data Collection

This study employed an exploratory qualitative interview approach to investigate how software engineering practitioners concretely constrain, supervise, and govern AI-assisted development workflows. Semi-structured interviews focused on AI-assisted coding workflows, code review practices, externalization, executable guardrails, and human oversight.

3.2 Participants

We conducted semi-structured interviews with five practitioners selected from a broader practitioner survey (Section 3.3), intentionally including a mix of roles (engineers, architects, technical leads, engineering managers) and organizational contexts (small product companies, large enterprises, agencies, industrial firms). Table 1 summarizes the participants; we refer to them as P1–P5 throughout. The interviews were intended to provide in-depth insight into emerging supervision practices rather than statistically representative coverage.

■ **Table 1** Interview participants and organizational contexts. Team size and governance posture are reported from each participant’s survey response.

ID	Role	Domain	Team size	Governance posture
P1	Software Architect / CTO	Enterprise software	Small (2–9)	Informal, AI-recommended
P2	Staff Engineering Manager	Energy utility frontend	Large (10+)	Clear guidelines
P3	Engineering Team Lead	Construction software	Large (10+)	Clear guidelines
P4	Senior Frontend Engineer	Digital agency	Small (2–9)	Informal arrangements
P5	Senior UI Engineer	Industrial technology	Small (2–9)	No explicit rules

3.3 Survey Context

To situate the interview findings, we draw on aggregate responses from a survey conducted in November–December 2025 within our institution’s computer science alumni network. The survey was co-developed by the first author and a colleague based on practical experience with AI-assisted tools, reviewed by one external practitioner for clarity, but not formally piloted. Approximately 100 alumni were personally invited; 50 completed the survey. Respondents were predominantly senior software engineers (16/50) and technical leads or architects (26/50), working mostly in small teams of 2–9 (34/50). Because the survey relied on convenience sampling and an unpiloted instrument, we do not interpret its aggregate responses as inferential evidence; we use them to provide indicative context for the qualitative findings.

3.4 Data Analysis

The interview material was analyzed using an iterative thematic analysis approach following the six-phase framework of Braun and Clarke [4]. Analysis began with close reading of interview transcripts and follow-up email responses, followed by open coding focused on recurring practices, tensions, supervision mechanisms, and organizational responses. Codes were iteratively refined into a scheme of six top-level categories (reconfiguration of validation, transformation of code review, understanding and explainability, guardrails and AI control mechanisms, context and maturity, and risks and failure modes), with three to eleven subcodes each; the full scheme is available in the supplementary material. Findings were continuously mapped back to supporting interview excerpts to strengthen traceability.

AI-supported analysis

The analysis was conducted by a single researcher (the first author), with ChatGPT used under researcher direction for auxiliary tasks: translation of German excerpts into English analytical paraphrases, surfacing recurring phrasing to support code consolidation, and drafting preliminary thematic summaries. All coding decisions, theme construction, evidence-to-theme mapping, and quote selection were made and verified by the researcher against the original transcripts. For the camera-ready revision, Claude (Anthropic) was used under the same direction to re-verify all directly quoted material against the original transcripts and correct several participant-attribution errors this process identified; all resulting changes were reviewed and approved by the first author.

Member-checking session

After the initial round of coding and thematic synthesis, we conducted a structured member-checking session with four of the five interview participants (P1, P2, P4, P5; P3 was unavailable) in late March 2026, in which preliminary findings were presented and discussed. Participants confirmed, qualified, or contradicted specific claims, and contributed additional examples and clarifications. The session was audio-recorded and transcribed with participant consent. Material from the session was incorporated into the final analysis: in particular, several illustrative examples and reframings reported in Section 4 and Section 5.5 originate from the member-checking session rather than from the original interviews.

Limitations of the analytical approach

Single-coder qualitative analysis without independent inter-rater reliability assessment introduces interpretive risk, which we partly mitigated through iterative refinement, written follow-up clarifications with each participant, and the member-checking session described above. We further discuss the limitations of the study, including sample size and the situated nature of the findings, in Section 5.7.

4 Findings

We report findings as five themes (F1–F5). Each quoted phrase is attributed to its participant; “participants” is reserved for statements supported by all five.

4.1 F1: Review-centric guardrails no longer scale, and specification quality becomes consequential

Participants consistently described AI-assisted development as substantially accelerating implementation while leaving review and validation constrained by human contextual reasoning. As P2 emphasized, code writing has become “*very cheap*” [P2] under AI-assisted development, with “*the bottleneck clearly moving to review*” [P2]. P1 quantified this concretely: manual review of an AI-assisted one-shot implementation typically takes “*three to four times as long*” [P1] as the generation itself (Section 4.2) – a ratio between the review and generation phases of the AI-assisted workflow, not a before/after-AI comparison. P1, P2, P4, and P5 described AI-assisted development as intensifying rather than reducing pressure on review-centric practices. For context, code quality and long-term maintainability were the most frequently selected risks in the survey (43 and 40 of 50 respondents respectively).

P1 observed that what has changed is not the existence of upstream specification work, which experienced teams have long invested in [8], but its *consequentiality*: specification quality now propagates directly into generated code, with weak input producing weak output that LLMs do not push back against the way an experienced engineer would. This makes upstream effort more consequential, not merely more abundant.

A recurring concern involved generated solutions that appeared correct at first glance while remaining difficult to validate contextually – a phenomenon we refer to as *false correctness*. P2, P4, and P5 described generated artifacts as syntactically correct and locally coherent, yet problematic regarding architectural consistency, explainability, or long-term maintainability. P2 reported being unable to change parts of a system because he could no longer trace “*why they are now where they are*” [P2], and P5 summarized the pattern in words he had already used in his own written survey response before the interview: “*functionally correct,*

but nobody understands why” [P5]. False correctness shifted the burden of validation away from behavioral testing toward reasoning about architectural intent, contextual consistency, explainability, and long-term maintainability.

4.2 F2: Teams externalize implicit context into upstream guardrails

P4 and P5 described existing repositories, documentation, and architectural guidelines as often insufficient for AI-assisted generation: AI systems inferred incomplete, outdated, or locally plausible patterns without reliably capturing intended architectural direction. P5 further noted that different artifacts and implementation signals could contradict each other or evolve independently, creating ambiguity about authoritative guidance.

In response, participants described externalizing contextual intent more explicitly before generation. P1 estimated that *“two thirds of the time go into specification refinement”* [P1], and P4 confirmed a similar shift in his own team: substantially longer architectural planning at the start of a task, with correspondingly shorter time spent actually writing code [P4]. These practices also appeared selectively in the survey: 13/50 respondents reported using steering files such as `.cursor/config` or `claude.md`, and 6/50 reported structured prompt workflows (Requirements → Design → Tasks) – suggesting that externalization practices are emerging but not yet widespread.

P4 emphasized that once a recurring constraint was identified, it had to be handed off to automatic enforcement: *“If a rule is relevant, it must be enforced through linting”* [P4] – the point at which externalization (identifying and formalizing the constraint) meets executable guardrails (Section 4.3). P1 additionally described orchestration sub-agents that monitor for drift during generation (discussed further in Section 5.6). These practices convert tacit knowledge into reusable *preventive* guardrails that shape generation upstream, rather than only *detective* guardrails that catch problems downstream. P4 and P5 noted important limits, however: not all architectural assumptions or organizational conventions can be formalized in advance, which is why teams complemented these upstream guardrails with executable ones embedded in the build (Section 4.3).

4.3 F3: Executable guardrails expand in importance

P1, P2, P4, and P5 described linting, unit testing, CI/CD pipelines, and automated validation as increasingly critical guardrails for AI-assisted workflows. P1 and P2 reported extending existing executable enforcement with additional checks detecting architectural violations, code duplication, dependency misuse, and deviations from organizational implementation constraints. P2 and P4 described how such executable enforcement made the build system itself the arbiter: violations of encoded rules would break the build rather than being caught by human reviewers. P5 illustrated the deliberate character of this stance in his own monorepo: he consciously chose not to relax automated conventions even though doing so would have been easy – *“as long as I can do it automatically it costs me nothing... I really want this convention to be strictly upheld”* [P5]. These executable guardrails were not limited to syntactic correctness but extended toward architectural conformance and organizational implementation policies.

P4 emphasized important limits: contextual interpretation, architectural tradeoff reasoning, and long-term maintainability assessment frequently could not be checked automatically. Executable guardrails thus carried part of the supervision load but did not replace human oversight, reducing recurring review burden while leaving contextual and interpretive assessment unresolved.

4.4 F4: Human oversight shifts toward supervisory and operational explainability

Despite increasing reliance on preventive and executable guardrails, participants consistently rejected fully autonomous workflows. P1, P2, and P4 emphasized that AI-generated artifacts continued to require a human guardrail layer in situations involving architectural tradeoffs, contextual ambiguity, or organizational constraints that could not be encoded reliably. As P1 summarized, AI-assisted development was “*no fire-and-forget*” [P1].

P1 described the human guardrail as more supervisory and intervention-oriented: rather than reviewing artifacts only after implementation completion, developers “*observe step-by-step... interrupt early*” [P1] when generation diverged from intended direction. Here intervention is an act of supervision triggered when a guardrail – whether an automated drift-detector or the observing developer – surfaces a problem. P4 additionally emphasized that the human guardrail focused on architectural reasoning and long-term maintainability rather than complete line-by-line comprehension. In place of full comprehension, P4 described *operational explainability*: the requirement that generated systems remain diagnosable and reconstructable on demand, supported by abstraction layers, visualization mechanisms, logs, and higher-level interpretive tooling that make the human guardrail effective without line-by-line reading.

The strength of the explainability requirement varied by stakes. P1 noted that organizational tolerance differed sharply by use case: a short-lived UX prototype may not require any line-level developer understanding, whereas a database migration or a feature in a system with strict recovery-time requirements demands precise comprehension of every statement. Operational explainability is therefore not a single threshold but a context-dependent capability calibrated to system criticality and longevity.

P2 and P4 raised concerns about the sustainability of such interpretive oversight, particularly for less experienced developers, a concern recently elevated to a broader profession-level question by Russinovich and Hanselman [15] (see also Section 5.7). Survey responses indicated related concerns: 23/50 respondents wished for targeted support on review standards for AI-generated code, and 15/50 on monitoring and traceability tools.

4.5 F5: Guardrail configurations vary along identifiable dimensions

The five participants varied substantially in how they combined the three guardrail layers – preventive guardrails, executable guardrails, and human oversight. We identify three dimensions along which configurations differed: (i) *governance posture* – the presence and formality of organizational AI-tool guidelines; (ii) *system criticality and homogeneity* – the longevity and consistency of the codebase under AI-assisted modification; and (iii) *team composition* – the distribution of senior and less experienced developers and the resulting supervisory capacity. We deliberately avoid the term “maturity” here, since it suggests a linear progression that our data does not support.

Along the governance dimension, P2 and P3 described environments with clear, formal AI-tool guidelines and structured review procedures, whereas P1, P4, and P5 reported predominantly informal arrangements or no explicit rules. Survey responses suggest similar heterogeneity beyond the interview sample: asked how AI-tool use is governed or monitored, 24 of the 50 respondents reported clear organizational guidelines and 23 reported only informal arrangements or no monitoring at all. Asked separately how collaboration had changed, 8 respondents reported that responsibility for AI-assisted code had become more diffuse.

System criticality and homogeneity strongly shaped which mechanisms worked. P1 explicitly framed his team’s practices as contingent on a modern, low-legacy, highly conformant codebase, and was explicit that the same mechanisms would likely not transfer to grown legacy systems. P5 correspondingly described legacy repositories as particularly challenging because inconsistent patterns and undocumented assumptions increased ambiguity.

P2 and P5 observed that teams with stronger architectural expertise used executable constraints more extensively. The lower bound is worth naming: teams that adopt AI tools widely without proportionate senior supervisory capacity may accumulate risk not visible from within the team, a concern consistent with broader observations on AI’s seniority-biased effects [15].

5 Discussion

5.1 From Review-Centric to Layered Supervision

Participants responded to the pressures identified in Section 4 by distributing supervision across three guardrail layers: preventive guardrails, executable guardrails, and human oversight. No single layer proved sufficient: preventive guardrails cannot capture all architectural tradeoffs in advance, executable guardrails cannot evaluate contextual appropriateness, and the human guardrail cannot scale to AI-generation throughput.

5.2 Pressure Toward Upstream Externalization

Traditional artifacts such as repositories, documentation, and architectural guidelines relied on human interpretation to be useful; in AI-assisted workflows, we read our data as indicating that this interpretive layer is largely absent in the traditional sense – not because AI systems process context without any interpretive mechanism at all (a probabilistic model plausibly performs a form of interpretation of its own), but because the human sense-making step that traditionally connected a written artifact to a considered action is no longer a reliable intermediary. Specification refinement, architectural planning, orchestration workflows, and the preventive guardrails they produce therefore take on a new role: they shape permissible generation trajectories ahead of implementation rather than only documenting decisions already made. Externalization does not eliminate the need for supervision, however: contextual assumptions, architectural tradeoffs, and organizational expectations can only ever be partially formalized, and it is often unclear in advance whether a given level of externalization is sufficient to reliably steer AI-assisted generation.

A natural question is where externalization ends and executable guardrails (Section 4.3) begin, since both act on generation before a human reviews the output. We distinguish the two by mechanism rather than by sequence. Externalization produces preventive guardrails – specifications, steering files, architectural plans – that guide what an AI system generates but are not themselves automatically checked; their effect depends on whether the generation process actually consults them. Executable guardrails, by contrast, are automatically evaluated against generated output regardless of how that output was produced. In practice the two run concurrently rather than as sequential stages: a steering file (a preventive guardrail produced through externalization) encoding a convention and a linting rule enforcing the same convention (an executable guardrail) typically coexist, so that the executable check still catches violations when the steering artifact is outdated, ignored, or absent from a particular generation trajectory. Externalization thus shapes generation *ex ante*; executable guardrails verify it *ex post*, independent of whether the ex-ante shaping succeeded.

5.3 What Changes Relative to Pre-LLM Software Engineering Governance

The individual mechanisms participants described – linting, CI/CD-based enforcement, architectural fitness functions, policy-as-code, specification-driven workflows – are not in themselves new. Each has a substantial pre-LLM history in software engineering practice and research [10, 16, 7, 8]. Our findings suggest that what is changing is not the existence of these mechanisms, but their operational role, relative weight, and configuration.

We highlight three shifts. Guardrails move *upstream*: steering files, structured prompts, and specification refinement increasingly act as *preventive* guardrails that guide generation, rather than the field’s traditional reliance on *detective* guardrails that inspect finished code (cf. Section 4.2). Executable guardrails are *promoted in priority*: P4 articulated a stance in which any sufficiently relevant rule is treated as a candidate for automatic linting enforcement (Section 4.2), reserving the human guardrail for what cannot be encoded (Section 4.3). Human oversight is *redistributed temporally*: P1 described monitoring generation step by step and intervening early [P1], shifting the human guardrail from post-hoc inspection toward concurrent supervision (Section 4.4). These shifts amplify and reconfigure existing practices under AI-induced throughput and contextual-opacity pressures, rather than replacing them.

5.4 Human Oversight Shifts from Inspection to Supervisory Interpretation

The reframing of oversight described in Section 4.4 carries an implication for expertise. Effective supervisory interpretation, abstraction-based assessment, and operational explainability all rest on architectural and contextual judgment that participants framed as in short supply. AI-assisted development may therefore shift professional expertise toward architectural reasoning, contextual interpretation, and supervisory intervention, with corresponding implications for how less experienced developers acquire this expertise – a profession-level question we return to in Section 5.6.

5.5 Risk–Productivity Trade-offs in Practice

The adoption of AI-assisted development introduces a tension between productivity gains and emerging validation costs. Participants described faster implementation and reduced effort in code-writing phases as genuine and substantial. As P4 emphasized during the member-checking session, organizations would not currently be investing this level of attention if the productivity gains were marginal – an inference we draw from participants’ observed investment of time and organizational effort rather than a claim any participant stated directly in these terms. At the same time, these gains were highly contingent on guardrails. Without sufficient preventive, executable, and human guardrails, the gains were frequently offset by increased review effort, accumulated false correctness, and delayed defect detection.

P2 recounted one such trajectory: in a project where AI-generated changes accumulated over several months without proportionate review capacity, a substantive feature had to be discarded and reimplemented from scratch once its architectural problems became visible. P2 framed this as a failure mode rather than an inherent property of AI-assisted development: “*had this been done without an AI system, we would not have generated so much code . . . maybe we would have noticed earlier*” [P2]. The lesson is not that AI-assisted generation is unsafe, but that productivity gains and validation infrastructure need to be scaled together – an organizational choice rather than a purely technical optimization.

5.6 Implications for Software Engineering Practice

The participants' practices suggest several concrete patterns rather than recommendations, since they emerged in specific organizational contexts and require adaptation rather than direct transfer.

Pattern 1: Promote recurring review findings into linting rules

P4 articulated a stance in which review effort is reserved for what cannot be encoded automatically (quoted in full in Section 4.2); P5 exemplified the operational consequence with the same deliberate strictness: *"I could leave it out – but I consciously chose not to. I really want this convention to be strictly upheld"* [P5]. The pattern is to treat each manually-recurring review concern as a candidate for promotion into an executable check, narrowing manual review toward the genuinely contextual. This aligns with established policy-as-code [7] and architectural fitness-function practice [8], applied as an explicit response to AI-generation throughput.

Pattern 2: Front-load specification and architectural planning

P1 estimated that *"two thirds of the time go into specification refinement"* [P1], with P4 reporting longer architectural planning at the start of a task and considerably shorter coding time [P4]. Effort that previously sat in implementation has migrated upstream; organizations should budget for this redistribution rather than treating planning as overhead to compress.

Pattern 3: Use sub-agents for concurrent supervision and independent review

Two complementary mechanisms emerged in P1's team. First, planner sub-agents constrain generation upstream: P1 described using *"three different planner sub-agents"* [P1] whose proposals are ranked by a further agent, with the explicit purpose of detecting when *"an agent has completely drifted"* [P1] from the intended plan – decoupling architectural-conformance checking from post-hoc review by embedding it concurrently with generation.

Pattern 4: Shift validation toward higher-level abstractions

In some teams, the primary response to AI-generated code volume is not deeper line-by-line inspection but *validation through higher-level artifacts* – architecture plans, visual system overviews, sequence diagrams, and other intermediate representations that make large changes easier to reason about and verify. P4 framed this as particularly important in agency contexts with heterogeneous projects and limited review capacity, where teams shift effort from code-level toward architecture-level validation, sometimes using AI-generated visual overviews to check intended structure, change scope, and system fit during implementation. Returns on tooling that makes architectural state and implementation trajectories *legible at a glance* appear larger than returns on tooling that merely accelerates generation.

A note on expertise

Across all patterns, effective application appeared to depend on architectural and contextual expertise that participants framed as in short supply (P2, P4). Whether organizations can sustain this expertise as implementation activities become increasingly automated is a broader question beyond the scope of this paper; we refer interested readers to recent profession-level discussions of the issue [15].

5.7 Limitations and Future Work

The findings are based on five interviews drawn from a single practitioner survey conducted within an alumni network through convenience sampling and an unpiloted instrument (Section 3.3); the practitioner population over-represents Swiss and Central European contexts, and practices in this space evolve rapidly. The analysis was conducted by a single researcher with AI-supported auxiliary work; we mitigated single-coder risk through iterative refinement, follow-up clarifications, and the member-checking session described in Section 3.4, though no independent inter-rater reliability assessment was performed.

One of the five interview participants (P4) is also a co-author of this paper, contributing the practitioner perspective required by the SEIP track. This dual role creates two distinct risks of bias. First, P4’s own statements could have been selected or framed more favorably than other participants’ during quote selection and writing. Second, P4’s involvement in the project could have reshaped the coding scheme or thematic framing around his own contributions rather than the full dataset. We mitigated the first risk by having the first author verify all quotations attributed to P4 against the original interview and member-checking transcripts independently, without P4’s input on selection or wording. We mitigated the second by finalizing the coding scheme and thematic framing before P4 joined as co-author, so that his subsequent contribution was limited to reviewing and co-writing the manuscript rather than to re-deriving the analysis.

The study captures reported practices and perceptions rather than direct longitudinal observation. Future work could complement interview-based findings with ethnographic observations, repository analyses (cf. [12, 1]), or longitudinal studies; cost-economic drivers of AI adoption, identified by participants as primary motivations but outside our analytic scope, also warrant investigation. Due to confidentiality agreements, full interview transcripts cannot be shared; the coding scheme, survey instrument, and anonymized evidence table are openly available in our supplementary Zenodo repository (<https://doi.org/10.5281/zenodo.21611622>).

6 Conclusion

AI-assisted development appears not to automate implementation in isolation but to redistribute supervision across the development process. Based on a qualitative interview study with five practitioners and a member-checking session, we identified a transition from review-centric guardrails toward *layered supervision*, in which the supervision function is distributed across three guardrail layers: preventive guardrails, executable guardrails, and human oversight. Existing engineering infrastructure – linting, testing, CI/CD, architectural validation – gains a new role as scalable supervision infrastructure for AI-assisted generation, while the human guardrail shifts from line-by-line inspection toward supervisory interpretation focused on architectural reasoning, explainability, and long-term maintainability.

References

- 1 Shyam Agarwal, Hao He, and Bogdan Vasilescu. AI IDEs or Autonomous Agents? Measuring the Impact of Coding Agents on Software Development. In *Proceedings of the 23rd International Conference on Mining Software Repositories (MSR)*, 2026. arXiv:2601.13597. doi:10.48550/arXiv.2601.13597.
- 2 Birgitta Böckeler. Harness Engineering for Coding Agent Users. martinowler.com, 2026. Part of Martin Fowler’s “Exploring Generative AI” series; published 2 April 2026, accessed 30 July 2026. URL: <https://martinfowler.com/articles/harness-engineering.html>.
- 3 Markus Borg, Dave Hewett, Nadim Hagatulah, Noric Couderc, Emma Söderberg, Donald Graham, Uttam Kini, and Dave Farley. Echoes of AI: Investigating the downstream effects of AI assistants on software maintainability. *Empirical Software Engineering*, 31, 2026. Article number 161. doi:10.1007/s10664-026-10889-1.

- 4 Virginia Braun and Victoria Clarke. Using thematic analysis in psychology. *Qualitative Research in Psychology*, 3(2):77–101, 2006. doi:10.1191/1478088706qp063oa.
- 5 DORA, Google Cloud. 2025 State of AI-Assisted Software Development. https://services.google.com/fh/files/misc/2025_state_of_ai_assisted_software_development.pdf, 2025. DevOps Research and Assessment (DORA) Report.
- 6 Melanie Fink. Human Oversight under Article 14 of the EU AI Act. *SSRN Electronic Journal*, 2025. Forthcoming in: *The AI Act Commentary: A Thematic Analysis*, Hart-Bloomsbury, 2026. doi:10.2139/ssrn.5147196.
- 7 Patrick Loic Foalem, Foutse Khomh, Leuson Da Silva, and Ettore Merlo. An Empirical Study of Policy-as-Code Adoption in Open-Source Software Projects. *arXiv preprint arXiv:2601.05555*, 2026. doi:10.48550/arXiv.2601.05555.
- 8 Neal Ford and Mark Richards. *Architecture as Code: Quantifying the Intersections of Software Architecture*. O'Reilly Media, 2027. Early Release.
- 9 Matthias Galster, Seyedmoein Mohsenimofidi, Jai Lal Lulla, Muhammad Auwal Abubakar, Christoph Treude, and Sebastian Baltes. Configuring Agentic AI Coding Tools: An Exploratory Study. In *Proceedings of the 3rd ACM International Conference on AI-Powered Software (AIware)*, 2026. doi:10.1145/3805760.3814887.
- 10 Marian Jureczko, Łukasz Kajda, and Paweł Górecki. Code review effectiveness: an empirical study on selected factors influence. *IET Software*, 14(7):794–805, 2020. doi:10.1049/iet-sen.2020.0134.
- 11 Jan H. Klemmer, Stefan Albert Horstmann, Nikhil Patnaik, Cordelia Ludden, Cordell Burton Jr., Carson Powers, Fabio Massacci, Akond Rahman, Daniel Votipka, Heather Richter Lipford, Awais Rashid, Alena Naiakshina, and Sascha Fahl. Using AI Assistants in Software Development: A Qualitative Study on Security Practices and Concerns. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security (CCS '24)*, pages 2726–2740, 2024. doi:10.1145/3658644.3690283.
- 12 Yue Liu, Ratnadira Widayarsi, Yanjie Zhao, Ivana Clairine Irsan, Junkai Chen, and David Lo. Debt Behind the AI Boom: A Large-Scale Empirical Study of AI-Generated Code in the Wild. *arXiv preprint arXiv:2603.28592*, 2026. doi:10.48550/arXiv.2603.28592.
- 13 Addy Osmani. Loop Engineering. [addyosmani.com](https://addyosmani.com/blog/loop-engineering/), 2026. Published 7 June 2026, accessed 30 July 2026. URL: <https://addyosmani.com/blog/loop-engineering/>.
- 14 Sida Peng, Eirini Kalliamvakou, Peter Cihon, and Mert Demirer. The Impact of AI on Developer Productivity: Evidence from GitHub Copilot. *arXiv preprint arXiv:2302.06590*, 2023. doi:10.48550/arXiv.2302.06590.
- 15 Mark Russinovich and Scott Hanselman. Redefining the Software Engineering Profession for AI. *Communications of the ACM*, 69(4), 2026. Opinion column. doi:10.1145/3779312.
- 16 Davide Spadini, Fabio Palomba, Tobias Baum, Stefan Hanenberg, Magiel Bruntink, and Alberto Bacchelli. Test-Driven Code Review: An Empirical Study. In *Proceedings of the 41st International Conference on Software Engineering (ICSE)*, pages 1061–1072, 2019. doi:10.1109/ICSE.2019.00110.
- 17 Ningzhi Tang, Meng Chen, Zheng Ning, Aakash Bansal, Yu Huang, Collin McMillan, and Toby Jia-Jun Li. An Empirical Study of Developer Behaviors for Validating and Repairing AI-Generated Code. In *Proceedings of the 13th Annual Workshop on the Intersection of HCI and PL (PLATEAU)*, 2023.
- 18 Maja Vukovic, Rangeet Pan, Tin Kam Ho, Rahul Krishna, Raju Pavuluri, and Michele Merler. Usage, Effects and Requirements for AI Coding Assistants in the Enterprise: An Empirical Study. In *Proceedings of the 3rd International Workshop on Large Language Models for Code (LLM4Code), co-located with ICSE 2026*, 2026. doi:10.1145/3786181.3788727.
- 19 Beiqi Zhang, Peng Liang, Xiyu Zhou, Aakash Ahmad, and Muhammad Waseem. Practices and Challenges of Using GitHub Copilot: An Empirical Study. In *Proceedings of the 35th International Conference on Software Engineering and Knowledge Engineering (SEKE)*, pages 124–129, 2023. arXiv:2303.08733. doi:10.48550/arXiv.2303.08733.