The End of Code Review: Coding Agents Supersede Human Inspection

Martin Monperrus
KTH Royal Institute of Technology
Stockholm, Sweden
monperrus@kth.se

***Abstract*—Code review has been the primary quality gate in software development since Fagan formalised code inspection in 1976. For five decades, having a human examine and comment on a colleague's changes before merge has been a cornerstone practice at organisations of every size. Coding agents are large language model (LLM)-based autonomous systems capable of reading, writing, testing, and repairing software. We argue that coding agents have crossed a threshold of capability at which traditional human code review is no longer a necessary component of a software quality pipeline. Our argument rests on three claims: every stated goal of code review can be served by agents at lower cost and higher throughput; the naive integration in which agents write code and humans remain the mandatory reviewers is a dead end because it neither provides meaningful assurance nor scales with AI-assisted throughput; and the cost-benefit balance of mandatory human inspection has already flipped for the majority of routine changes. We synthesise evidence from benchmark studies and industry deployments and draw out implications for workflows, tooling, team structure, open-source dynamics, and the research agenda the software engineering community must pursue to govern this transition responsibly.**

I. Introduction

Code review is the dominant human quality gate in modern software development. Since Bacchelli and Bird surveyed its practice at Microsoft [1] and Sadowski et al. documented it at Google [2], the field has understood code review as serving four overlapping goals: finding defects before they reach production, enforcing style and conventions, transferring knowledge between team members, and building shared awareness of the evolving codebase. No serious software team omits it.

Yet code review carries substantial costs that are often underappreciated. Developers at large organisations spend between ten and fifteen percent of their working hours reading and commenting on others' code [2]. The review latency between submitting a pull request and receiving actionable feedback routinely stretches over twenty-four hours and can extend to days, imposing a structural drag on continuous-delivery [3]. Beyond time, review generates social friction: tone escalations, seniority bias, and the well-documented tendency for first-time contributors to abandon projects after critical feedback [4], [5].

Into this landscape, coding agents have arrived. LLM-based systems such as Claude Code, Codex, and GitHub Copilot can now read and modify files, execute test suites, interpret compiler output, and iteratively repair failures without human direction [6]–[9]. On SWE-bench (a benchmark of real, unmodified GitHub issues drawn from popular Python libraries), state-of-the-art agents resolve more than eighty percent of tasks end-to-end [10]. Work on LLM-based code review shows that agents can produce inline defect comments at quality comparable to trained human reviewers [11], [12].

In this paper, we make a strong claim: *coding agents have reached a capability threshold at which human code review is redundant and should be replaced by agent-driven verification.* We do not present a new empirical study; instead, we synthesise existing capability evidence, articulate a three-part structural argument for transition, and enumerate the implications for software engineering practice, tooling, and research. Specifically, we argue that agents can satisfy the established goals of review, that the intermediate model in which agents write code but humans remain mandatory reviewers is unstable, and that the economics of mandatory human inspection have already turned negative for routine changes.

To sum up, our contributions are:

- The demonstration that every stated goal of code review—defect detection, style enforcement, knowledge transfer, and team awareness—can be met by coding agents at lower cost and higher throughput than human reviewers.
- The argument that combining AI code generation with mandatory human review is not a stable endpoint: it creates the appearance of assurance while turning review capacity into the next delivery bottleneck.
- The cost-benefit analysis showing that mandatory human inspection has already flipped; agent reviews are instantaneous, consistent, and auditable, while the marginal defect-detection value of human review shrinks as agent capabilities grow.

Section II provides background on code review, coding agents, and prior automated review work. Section III surveys the evidence for agent capability. Section IV develops the core argument across three claims. Section V draws out implications for SE practices, tooling, and future research directions. Section VI presents counter-arguments and rebuttals. Section VII concludes.

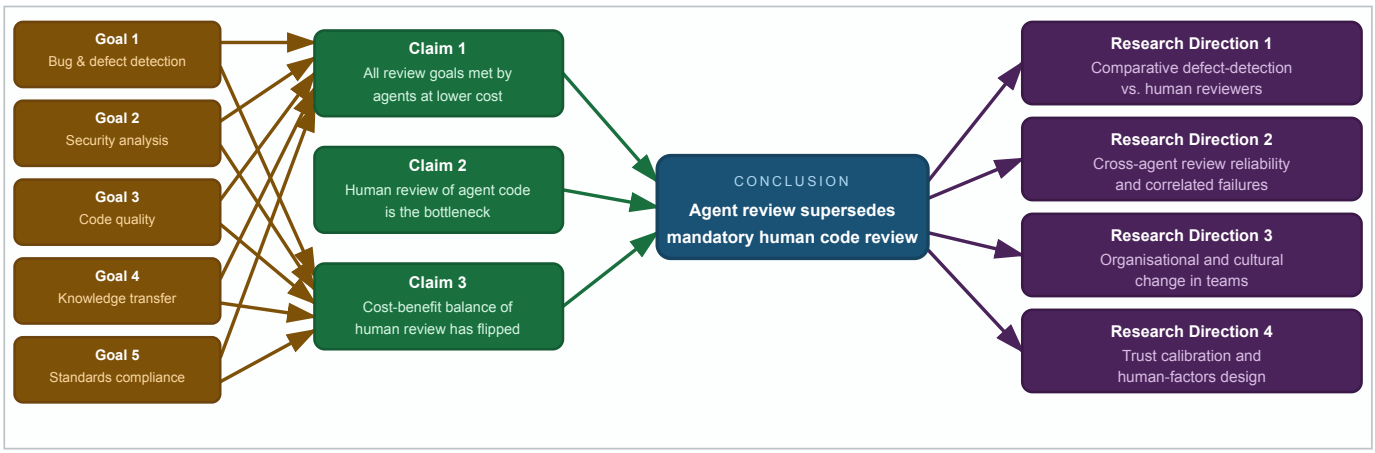

Fig. 1. Argument map of the paper. Agentic code review reshapes software development and creates research opportunities.

II. Background

A. Code Review: History and Practice

Code review as a formal engineering discipline originates with Fagan's 1976 paper on code inspections at IBM [13]. Fagan proposed a structured, multi-stage process: planning, overview, preparation, inspection, rework, and follow-up. The process was presented as expensive, requiring up to three percent of total project effort.

Over the following decades, the practice evolved substantially. Rigby and Bird documented a convergence across projects and organisations toward *lightweight* code review: asynchronous, tool-mediated inspection of small changesets, conducted by one or two reviewers rather than a formal panel [14]. The modern instantiation of this model is the pull request, introduced by distributed version-control platforms such as Github and now ubiquitous in both commercial and open-source software development.

The empirical record on the effectiveness of modern code review is more nuanced than the practice's universal adoption might suggest. Bacchelli and Bird found that defect detection, while a primary motivation, is not what reviewers most reliably deliver: style corrections, minor improvements, and questions about intent dominate the comment corpus [1]. Czerwonka et al. reached the pointed conclusion that code reviews at Microsoft "do not find bugs" in the sense of catching deep, logic-level defects; their primary value lies in knowledge transfer and maintainability improvement [15]. McIntosh et al. report a correlation between review coverage and post-release defect density [16], though correlation does not establish that it is the review, rather than team discipline more broadly, that drives quality. Knowledge transfer and socialisation emerge consistently across studies as valued outcomes of the review interaction [1], [4].

B. Automated Code Review

Efforts to automate code review predate large language models. Early approaches relied on pattern matching, rule-based checkers, and machine-learning classifiers trained on review-comment corpora to predict whether a change would attract a comment and what that comment would say. These systems automated individual review activities in isolation: comment prediction, priority ranking of changed files, detection of specific defect classes.

The LLM era opened new possibilities. CodeReviewer [11] pre-trains a model on a large corpus of pull-request diffs and associated review comments, achieving competitive performance on comment generation, code-refinement prediction, and review-necessity classification. LLaMA-Reviewer [17] applies parameter-efficient fine-tuning to adapt a general-purpose LLM for the review-comment generation task, demonstrating that strong review performance is achievable without full fine-tuning. Tufano et al. [18] build an end-to-end pipeline from diff to refined patch, showing that a model can both generate a review comment and produce the corrected code, eliminating one round-trip in the review-revise cycle. Pornprasit and Tantithamthavorn evaluate these and related systems in industrial settings and find meaningful coverage across defect types [12]. Tang et al. introduce CodeAgent [19], a multi-agent system in which specialised communicative agents collaborate to perform code review, pushing the automation boundary from single-model comment generation toward coordinated agent workflows. What these systems share is a focus on automating *activities within* a review workflow that remains human-governed; they improve the efficiency of specific steps without questioning whether the overall review gate is necessary.

A parallel strand considers bots in software repositories more broadly. Lebeuf et al. characterise the diversity of software bots from simple automation scripts to sophisticated conversational agents and their effects on developer workflows [20]. Wessel et al. study the power dynamics and community effects of bots in open-source projects, finding that bots accelerate certain activities but also create friction when developers perceive them as inadequately contextualised [21]. This work anticipates many of the adoption challenges that agent review will face.

To our knowledge, this paper is the first to argue for the

complete displacement of mandatory human code review by coding agents, and to enumerate the resulting implications for workflow, tooling, research, and standards in a unified treatment.

C. Coding Agents

A coding agent is a system in which a large language model (LLM) is embedded in an *agentic loop*: the model can invoke tools for reading and writing files, executing shell commands, running tests, querying documentation, and then iterate until a goal is achieved or a stopping condition is met. The LLM provides language understanding, code synthesis, and contextual reasoning; the tool loop provides grounding in the actual state of the software artefact and its execution.

Representative agents include OpenAI Codex [22], an interactive assistant built on top of the GPT line of models; Claude Code [23], an agentic coding assistant that operates directly in the developer's terminal; SWE-agent [6], which introduces an agent-computer interface optimised for repository-scale software engineering tasks; Devin [7], a commercial system that autonomously navigates codebases and deploys fixes; and GitHub Copilot [9], which integrates agent capabilities directly into the pull-request workflow.

III. Evidence of Agent Capability

To support the claim that coding agents can displace human code review, we survey evidence of agent capability across three dimensions: benchmark performance on software-engineering tasks, review-specific capabilities, and developer productivity with deployed tools.

A. Benchmark Performance

The most direct evidence for agent capability comes from SWE-bench, a benchmark that evaluates whether a system can resolve real, unmodified issues drawn from popular open-source Python projects [10]. Unlike synthetic coding challenges, SWE-bench tasks require understanding issue descriptions, navigating multi-file repositories, modifying the correct files, and producing patches that pass the project's existing test suite. Early results were sobering: GPT-4 with retrieval-based context selection resolved approximately 1.7 % of tasks. SWE-agent, by introducing an agent-computer interface that structures the model's interaction with the file system and shell, raised this figure to approximately 12.5 % [6]. By late 2024, the best-performing systems on SWE-bench Verified—a curated subset with verified ground-truth resolutions—exceeded 50 %. By late 2025, top agents on the public leaderboard resolved more than 70 % of tasks [24]. The improvement trajectory, from under two percent to over seventy percent in roughly two years, is without precedent in the history of automated software engineering tools.

Related evidence comes from automated program repair. Xia et al. demonstrate that LLM-based repair approaches substantially outperform earlier generate-and-validate systems like GenProg [25], [26], fixing a far larger proportion of benchmark bugs without requiring manually specified test cases or fix templates. At the competitive programming level, AlphaCode ranked within the top 54 % of human participants on Codeforces contests [27]. This establishes that LLMs can reason about non-trivial algorithmic problems.

B. Review-Specific Capabilities

Beyond general software engineering, several strands of work speak specifically to the capabilities that code review requires. Pornprasit and Tantithamthavorn evaluate LLM-based automated review in industrial settings and find that agents detect the same categories of defect that human reviewers target: correctness errors, security weaknesses, performance inefficiencies, and style violations [12]. Li et al. demonstrate that CodeReviewer produces actionable inline comments at quality that is at least comparable to those of trained human reviewers on a significant fraction of the evaluation set [11].

Agents bring capabilities that human reviewers structurally cannot match. A human reviewer reads a diff; an agent can simultaneously hold the full file, the complete test suite, the git history of every touched function, and the project's documentation in context. A human reviewer can identify a problem and describe it in a comment; an agent can identify the problem, generate a fix, apply it, run the tests, and close the review loop without any human scheduling. A human reviewer has a calendar, a timezone, and finite attention; an agent operates continuously, applying the same scrutiny at 3 am on a Sunday as during a Monday morning sprint. These are not incremental improvements; they are structural advantages that compound as the scale and velocity of software development increase.

C. Developer-Productivity Evidence

Productivity studies with deployed AI coding tools corroborate the trajectory. Peng et al. conducted a controlled experiment measuring the effect of GitHub Copilot on task completion time and found a 55 % speed increase for developers with access to the tool [28]. Ziegler et al. analysed acceptance rates of LLM code completions over time and observed that developers increasingly accept rather than reject machine-generated suggestions, indicating growing calibrated trust [29]. Taken together, these results suggest that developers are not merely tolerating AI assistance; they are already integrating it into their workflows in ways that change the nature of the work, which is a precondition for the deeper integration that agent-based review represents.

IV. The End of Code Review: The Argument

A. Claim 1: Every goal of code review can be served by agents at lower cost and higher throughput.

Bacchelli and Bird identify four primary goals of code review: defect detection, style and standards enforcement, knowledge transfer, and team awareness [1]. We argue that contemporary coding agents satisfy each goal at least as well as human reviewers, and in several dimensions markedly better.

Defect detection. Human reviewers are effective at identifying surface-level issues but unreliable at catching deep logical bugs [15]. **Agents powered by large language models, by contrast, can perform exhaustive dataflow reasoning, cross-reference the full test suite, and apply learned patterns from millions of open-source repositories.** Recent work shows that LLM-based review systems produce actionable defect reports comparable in quality to those of trained human reviewers, while operating on every commit without fatigue or time-zone constraints [11], [12].

Style and standards enforcement. Linters, formatters, and type checkers have long handled syntactic and stylistic concerns automatically. **Agents extend this capacity to semantic style: naming consistency, idiomatic API usage, and documentation conventions can all be enforced through LLM-based rewriting, eliminating an entire category of reviewer comment.**

Security. Security review is precisely the domain where human attention is most overloaded and most consequential. Pearce et al. demonstrate that GitHub Copilot introduces common weakness enumeration (CWE) violations at non-trivial rates [30], yet the same generation capability, when redirected toward *detection*, allows agents to enumerate vulnerability classes far more systematically than a developer performing an ad-hoc review [31]. **AI-based security scanners already outperform many manual reviewers on standard vulnerability benchmarks [32]**.

Knowledge transfer. Knowledge transfer through code review has always been a by-product rather than a designed mechanism: a reviewer who happens to understand an unfamiliar module provides context in a comment. **Agents can actively generate on-demand explanations, architectural summaries, and updated documentation at merge time, which is a more reliable and scalable mechanism for propagating knowledge than the incidental commentary of a busy colleague.**

B. Claim 2: The naive integration—AI coding with human review—is a dead end.

The first response of most organisations to the rise of AI coding tools is as follows: an agent writes the code, a human reviews it. This arrangement feels conservative and safe. We argue that it is neither, and that it fails on two independent grounds.

Human review provides no genuine assurance. The traditional rationale for code review rests on a tacit assumption: human developers write code, and human reviewers provide an independent check. When an agent generates the code, this assumption collapses. A large language model can produce hundreds of lines of plausible, internally consistent code that contains subtle semantic errors—errors that are invisible without running the full test suite or performing the kind of exhaustive dataflow analysis that only an agent can do systematically. Human reviewers, reading a diff on a screen, are poorly positioned to catch the errors that AI-generated code may contain. In practice, reviews of agent-generated code become rubber-stamps: the human approves because the code looks correct, because the tests pass, and because the cognitive cost of genuine scrutiny is prohibitive [15]. The assurance that review is supposed to provide is illusory.

Human review does not scale. AI coding tools increase developer throughput. A developer assisted by an agent produces more commits, more pull requests, and more lines of changed code per day than an unassisted developer [28]. Human review capacity does not scale and the result is a bottleneck that grows proportionally with the productivity gain that AI coding delivers. Organisations that deploy AI coding tools while retaining human-gated review will find that the review queue becomes the binding constraint on their delivery pipeline. The faster the agents write, the longer the queue grows, and the more review degrades into a formality performed under time pressure. Retaining human review in this context does not preserve quality; it discards both the productivity benefit of AI coding and the quality benefit of genuine review.

The logically consistent step is to close the loop: have an independent agent review agent-generated code. Under this model, the human role shifts from *inspector* to *approver of agent decisions*: a human reviews only the structured summary of what the agents agreed upon, intervening solely when the agents flag uncertainty or when a change crosses a risk threshold that warrants human judgment.

C. Claim 3: The cost-benefit calculation has flipped.

Code review imposes measurable costs: developers at large organisations spend 10–15 % of their working hours reviewing code [2], and review latency introduces delays into continuous-delivery pipelines [3], [33]. These costs were historically justified by the defects that review caught. The justification weakens as agent coverage grows.

Consider the marginal value of a human review comment. As agents scan every file, every test, and every commit with increasing recall, the set of defects that escape agent review but would be caught by a human reviewer shrinks. At the same time, the cost of human review in time, in latency, in social friction [4], [5], remains constant. **The crossover point, at which the marginal benefit of human review no longer justifies its marginal cost, has already been reached.** Agent reviews are instantaneous, deterministic, and auditable: they produce structured reports, not informal threaded comments, and they can be re-run at any point in the pipeline. The competitive advantage of continuous delivery [33] makes review latency an increasingly unacceptable tax on software teams, and agents eliminate it entirely.

V. Implications

A. Implications for Software Engineering Practices

Merge workflow redesign. The most immediate implication of agent-based review is the restructuring of the pull-request workflow. In the current model, a developer opens a pull request and waits, sometimes for days, for a colleague to find time to review it. We propose replacing this gating step with an *agent-in-the-loop* verification pipeline that runs

automatically on every candidate merge. Merge gates shift from a human approval checkbox to a structured agent sign-off: test coverage thresholds, security scan results, style compliance reports, and reasoning traces are all produced without human scheduling. Human approval is not eliminated; it is reserved for the decisions that genuinely require it: high-risk changes, novel architecture choices, and regulated code paths where a named human must bear legal accountability. For the vast majority of commits (incremental features, bug fixes, dependency updates, refactors) agent sign-off is sufficient, and requiring more is waste.

Team structure and roles. As agent review matures, the role of the professional code reviewer will diminish in its current form. The time developers spend in synchronous review meetings and asynchronous comment threads will shrink. This does not mean that developer expertise becomes less important; it means that expertise is redirected. Developers increasingly become *specifiers* and *orchestrators*: they articulate requirements precisely enough for agents to act on them, evaluate agent-produced artefacts at a higher level of abstraction, and intervene when agent reasoning is uncertain or the stakes are high. The craft shifts from line-by-line inspection to system-level judgment. This is a change that many developers will welcome, since surveys consistently show that reviewing code is among the least enjoyable parts of the job [2].

Onboarding and knowledge transfer. One of the most-cited benefits of code review is its role in propagating knowledge among team members, particularly for newcomers learning a codebase [1]. This benefit must not be discarded; it must be redesigned. Agents can generate on-demand explanations of any file, function, or architectural decision, keyed to the context of a specific change. A new contributor who triggers an agent review can receive an annotated, tailored summary of relevant conventions and prior decisions. This is, in some respects, richer than what a busy senior engineer could provide in a rushed review comment. The genuine risk is a reduction in the informal, bidirectional conversations through which tacit knowledge and team culture propagate. Organisations that adopt agent review at scale will need to invest in complementary mechanisms such as pair programming sessions, design discussions, structured mentorship, to preserve the social and epistemic cohesion that review once provided incidentally.

Accountability and traceability. Agent review produces structured, immutable artefacts (logs, reports, and reasoning traces) that are in many respects superior to the informal threaded comments that currently constitute the audit trail for a merged pull request. A SARIF-formatted agent review report records exactly which rules were applied, which lines were flagged, and what reasoning the agent produced, in a form that is machine-readable and archivable. This has implications for regulated industries, where auditors require evidence that changes were reviewed according to specific criteria. We anticipate that regulatory frameworks will evolve to recognise "agent review certificates" as valid evidence of due diligence, analogous to the way that passing a static-analysis clean build is already accepted as evidence of certain quality properties. At the same time, the liability model must be updated: when an agent approves a change that subsequently causes harm, the question of accountability deserves sustained attention from the software engineering and legal communities.

Open-source dynamics. In open-source projects, maintainer bandwidth is a structural bottleneck. Contributor pull requests can wait weeks or months for review; projects are abandoned not because contributors stop submitting patches but because maintainers stop having time to review them [21]. Agent review directly addresses this bottleneck: a project that deploys an agent reviewer can process contributions at the rate they arrive, providing immediate feedback to contributors regardless of timezone or maintainer availability. The likely result is a compression of the feedback loop that currently discourages first-time contributors and drains maintainer energy. If the cost of submission is perceived to be lower, marginal contributions to open-source may increase.

B. Implications for Tooling

CI/CD integration. Agent review is a natural extension of the continuous integration pipeline. Just as build systems and test runners execute automatically on every push, a review agent can be triggered at the same point, on every commit to a feature branch, or as a mandatory gate before merge. This reframes review from an asynchronous social activity into a first-class engineering check, analogous to compilation: it runs, it reports, and it either passes or blocks [33]. Agents produce structured reports (JSON, SARIF) that CI dashboards, security platforms, and analytics systems can consume without human mediation. Review history becomes an engineering artefact, versioned and queryable, rather than a thread of informal comments that ages poorly.

IDE and editor integration. The latency reduction that agents enable is not confined to the merge pipeline. An agent embedded in a developer's editor can review changes before they are committed: as the developer writes, the agent reads context, identifies issues, and surfaces them inline. This collapses the feedback loop from hours or days to seconds. The interaction model also changes: rather than reading a list of review comments and deciding whether to accept or dismiss each one, the developer converses with the agent in natural language, asking for explanations, requesting alternative implementations, or negotiating tradeoffs. This is closer to pair programming than to traditional review, and it operates without the social overhead that makes synchronous pairing difficult to sustain at scale.

Version control platforms. For agent review to function as a first-class activity, platforms such as GitHub and GitLab must extend their identity and permission models to support agent actors. An agent reviewer needs a cryptographically signed identity, a record in the platform's audit log, and the ability to approve, request changes, or block merge in the same way a human reviewer can [9]. The pull-request user interface must evolve accordingly: agent-generated summaries, confidence scores, categorised findings, and inline fix suggestions

should be presented as structured UI components, not as synthetic comment threads that mimic human behaviour. Review history in this model becomes machine-readable by design: downstream analytics can operate directly on agent review records (defect trend dashboards, security posture tracking, codebase health scoring) .

Need for New Metrics. The metrics by which software teams measure review health will require redesign. Traditional approaches measures review turnaround time, comment density, approval latency. They are meaningful only when review is a human activity. When agents review, turnaround time approaches zero by construction, and comment density is a function of agent configuration rather than reviewer engagement. New metrics are needed: *agent review recall*, defined as the fraction of seeded or independently confirmed defects that the agent detected; *fix rate*, the fraction of agent-identified issues resolved automatically without human action; and *escape rate*, the fraction of defects that reach production despite agent sign-off. These metrics connect agent review directly to quality outcomes, enabling organisations to calibrate agent configurations and escalation thresholds empirically rather than anecdotally.

C. Implications for Future Research

The transition to agent-based code review opens a rich set of empirical and design research questions that the software engineering community is uniquely positioned to address.

RQ1: Comparative defect-detection performance. What is the defect-detection recall and precision of state-of-the-art agents compared to human reviewers on a controlled, diverse benchmark? Such a study would require a carefully constructed dataset of changes with known ground-truth defects, sampled across defect types (logic errors, security weaknesses, performance regressions, style violations), programming languages, and application domains. Without this benchmark, claims about agent superiority or inferiority rest on proxy evidence.

RQ2: Cross-agent review reliability. Can an agent reliably detect defects in code generated by a *different* agent? This question is practically urgent: as a growing fraction of commits originate from coding agents, cross-agent review is the natural operational scenario. The key concern is correlated failure: do models from the same family share blind spots that a model from a different family would catch? Controlled experiments varying both generator and reviewer models are needed to characterise failure-mode correlation.

RQ3: Organisational and cultural change. What organisational and cultural changes accompany the removal of mandatory human code review? A longitudinal field study following teams that transition from human-gated to agent-gated review could measure developer satisfaction, team cohesion, and tacit knowledge preservation. This is the kind of research that only the software engineering community, with its mixed-methods expertise, is equipped to conduct.

RQ4: Trust calibration and human factors. How do developers calibrate trust in agent review output? What user interface and interaction designs promote appropriate reliance, neither uncritical acceptance nor reflexive dismissal? What factors cause developers to override agent decisions, and are those overrides beneficial or harmful?

VI. Discussion

A. Counter-argument: Hallucination and false negatives.

LLMs can miss defects that fall outside their training distribution [31]. A human reviewer who is unsure about a change will typically say so; an LLM may silently produce an approval. The primary mitigation is ensemble review: running multiple independent agents, potentially based on different models and prompting strategies, and requiring consensus before sign-off. A second mitigation is calibrated uncertainty reporting: reviewer agents should be trained and evaluated to abstain ("I don't know") and emit confidence estimates that track empirical correctness, rather than always issuing a binary approval [34].

B. Counter-argument: Security vulnerabilities in agent-generated code.

Pearce et al. demonstrate that GitHub Copilot introduces common weakness enumeration (CWE) violations at non-trivial rates across a range of security-relevant coding scenarios [30]. When the same model family that generates code is also responsible for reviewing it, there is a risk that generative and review blind spots are correlated: the agent may fail to flag the very pattern of vulnerability that its generation tendency produces. The mitigation is to use cyber-specialized frontier reviewers for security sign-off: recent benchmark-oriented reports show that the latest models can outperform traditional static analyzers and strong baselines on vulnerability-identification tasks [35], [36].

C. Counter-argument: Adversarial inputs and prompt injection.

A sophisticated adversary who can submit a pull request may also craft code that, when read by a reviewing agent, manipulates the agent's reasoning through embedded natural-language instructions. A maliciously crafted comment, identifier, or string literal could instruct the agent to overlook a vulnerability or to approve a change it should block. This is an active area of security research with no fully solved defences [37], and it represents a qualitatively new attack surface that does not exist for human reviewers. Organisations deploying agent review must treat agent reviewer prompt injection as a first-class threat model.

D. Counter-argument: Architectural coherence requires human judgment.

At the architectural level, reviewers assess whether a change is consistent with the system's long-term design: whether a new abstraction duplicates an existing one, whether a local trade-off accumulates strategic technical debt, or whether an interface decision forecloses future extension. These judgements depend on a mental model of the system's architecture that AI may not yet hold with sufficient fidelity [1].

This concern misidentifies the scope of what pull-request review actually delivers. Architectural coherence is best enforced through design documents, architecture decision records, and dedicated architecture reviews, not through per-commit diff inspection. Conflating the two overstates what human PR review reliably provides, since empirical studies show that reviewers overwhelmingly focus on surface defects and style rather than strategic architectural validity [1], [15].

E. Counter-argument: Ethical accountability requires human judgment.

At the ethical level, code changes can carry consequences for user privacy, algorithmic fairness, and environmental footprint that require values-based judgement rather than correctness verification. Agents are calibrated to optimise for technical quality metrics; they are not reliably equipped to detect that a telemetry change violates a user's reasonable privacy expectation or that a ranking modification amplifies demographic bias [4]. Legal and organisational accountability frameworks assume a named human decision-maker: an automated approval leaves an accountability gap when a change causes harm.

Agents already flag security-sensitive patterns (PII logging, insecure randomness, over-privileged API calls); their coverage of fairness and compliance properties is growing [30], [31]. More fundamentally, mandatory PR review is not the appropriate locus for ethical scrutiny of software: that responsibility belongs in requirements engineering, product decision processes, and post-deployment monitoring, not in a diff-level approval gate. The accountability question is addressed by the same governance frameworks that apply to other automated quality gates: CI pipelines block deploys, static-analysis tools gate releases, and automated scanners flag security regressions, none of which require a named human responsible for every individual verdict. Agent review can be governed by identical institutional mechanisms, with human escalation reserved for changes that agents flag as uncertain or that cross an explicit risk threshold.

VII. Conclusion

Code review has served software engineering for five decades. From Fagan's formal inspections [13] to the pull-request model that now governs millions of daily commits [1], having humans read each other's code before merge has been treated as a mandatory engineering virtue. We have argued that this assumption no longer holds.

Coding agents have reached a capability level at which every stated goal of code review can be met automatically, faster, and at a scale that human review cannot match. The transition is not hypothetical; it is already underway for routine changes in industry. Agents review dependency bumps, refactors, and test additions today, with humans providing oversight only at the margins. The open question is not whether this shift will occur, but how quickly it will extend from routine changes to the full breadth of software development, and how the profession will manage that extension.

The software engineering community must act proactively. Workflows need to be redesigned around agent-in-the-loop verification rather than adapted incrementally from human-review processes. Standards must be updated to specify what constitutes adequate agent review for compliance purposes. New tooling must provide the infrastructure of agent identities, structured report formats, and calibrated metrics in order to make agent review first-class. Human factors research must characterise how developers should interact with agent review output to maintain appropriate oversight without re-introducing the latency, friction and rubber-stamping that agent review eliminates.

Code review will not disappear overnight. Its role will refocus to a thin layer of high-stakes human oversight: architecture decisions with long-lived consequences, security-critical paths in regulated systems, and changes whose correctness depends on requirements that no agent has been given access to. For everything else, the case for mandatory human review has already weakened to the point where it is difficult to defend on technical grounds.

The end of code review, as we once thought the absolute best practice of modern software engineering, is the beginning of a more productive way to build software.

References

[1] A. Bacchelli and C. Bird, "Expectations, outcomes, and challenges of modern code review," in *Proceedings of the 35th International Conference on Software Engineering (ICSE)*. IEEE, 2013, pp. 712–721.

[2] C. Sadowski, E. Söderberg, L. Church, M. Sipko, and A. Bacchelli, "Modern code review: a case study at Google," in *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. ACM, 2018, pp. 181–190.

[3] M. Hilton, T. Tunnell, K. Huang, D. Marinov, and D. Dig, "Usage, costs, and benefits of continuous integration in open-source projects," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. ACM, 2016, pp. 426–437.

[4] A. Bosu, M. Greiler, and C. Bird, "Process aspects and social dynamics of contemporary code review," in *IEEE Transactions on Software Engineering*, vol. 43, no. 1. IEEE, 2016, pp. 56–75.

[5] A. Murgia, P. Tourani, B. Adams, and M. Ortu, "Do developers feel emotions? an exploratory analysis of emotions in software artifacts," in *Proceedings of the 11th Working Conference on Mining Software Repositories (MSR)*. ACM, 2014, pp. 262–271.

[6] J. Yang, C. E. Jimenez, A. Wettig, K. Lieret, S. Yao, K. Narasimhan, and O. Press, "SWE-agent: Agent-computer interfaces enable automated software engineering," *arXiv preprint arXiv:2405.15793*, 2024.

[7] Cognition AI, "Introducing Devin, the first AI software engineer," *Cognition AI Blog*, 2024, https://www.cognition.ai/blog/introducing-devin.

[8] X. Wang, B. Chen, Y. Yuan, Y. Zhang, B. Li, C. Qian *et al.*, "OpenDevin: An open platform for AI software developers as generalist agents," in *arXiv preprint arXiv:2407.16741*, 2024.

[9] GitHub, "GitHub Copilot Workspace: Welcome to the Copilot-native developer environment," *GitHub Blog*, 2024, https://github.blog/2024-04-29-github-copilot-workspace/.

[10] C. E. Jimenez, J. Yang, A. Wettig, S. Yao, K. Pei, O. Press, and K. Narasimhan, "SWE-bench: Can language models resolve real-GitHub issues?" *arXiv preprint arXiv:2310.06770*, 2023.

[11] Z. Li, S. Lu, D. Guo, N. Duan, S. Jannu, G. Jenks, D. Majumder, J. Green, N. Sundaresan, M. Fu *et al.*, "CodeReviewer: Pre-training for automating code review activities," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, 2022, pp. 1536–1546.

[12] C. Pornprasit and C. Tantithamthavorn, "Automated code review in practice," in *Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2023, pp. 394–405.

[13] M. E. Fagan, "Design and code inspections to reduce errors in program development," *IBM Systems Journal*, vol. 15, no. 3, pp. 182–211, 1976.

[14] P. C. Rigby and C. Bird, "Convergent contemporary software peer review practices," pp. 202–212, 2013.

[15] J. Czerwonka, M. Greiler, and J. Tilford, "Code reviews do not find bugs: How the current code review best practice slows us down," pp. 27–28, 2015.

[16] S. McIntosh, Y. Kamei, B. Adams, and A. E. Hassan, "The impact of code review coverage and code review participation on software quality," in *Proceedings of the 11th Working Conference on Mining Software Repositories (MSR)*. ACM, 2014, pp. 192–201.

[17] J. Lu, L. Yu, X. Li, L. Yang, and C. Zuo, "Llama-reviewer: Advancing code review automation with large language models through parameter-efficient fine-tuning," in *Proceedings of the 34th IEEE International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2023, pp. 647–658.

[18] R. Tufano, S. Masiero, A. Mastropaolo, L. Pascarella, D. Poshyvanyk, and G. Bavota, "Using pre-trained models to boost code review automation." ACM, 2022, pp. 1–12.

[19] X. Tang, K. Kim, Y. Song, C. Lothritz, B. Li, S. Ezzini, H. Tian, J. Klein, and T. F. Bissyande, "CodeAgent: Autonomous communicative agents for code review," *arXiv preprint arXiv:2402.02172*, 2024.

[20] C. Lebeuf, M.-A. Storey, and A. Zagalsky, "Software bots," *IEEE Software*, vol. 35, no. 1, pp. 18–23, 2018.

[21] M. Wessel, B. M. de Souza, I. Steinmacher, I. S. Wiese, I. Polato, A. P. Chaves, and M. A. Gerosa, "The power of bots: Characterizing and understanding bots in OSS projects," in *Proceedings of the ACM on Human-Computer Interaction (CSCW)*, vol. 2. ACM, 2018.

[22] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. d. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman *et al.*, "Evaluating large language models trained on code," *arXiv preprint arXiv:2107.03374*, 2021.

[23] Anthropic, "The Claude 3 model family: Opus, Sonnet, Haiku," *Anthropic Technical Report*, 2024.

[24] S. bench Team, "SWE-bench leaderboard," https://www.swebench.com, 2025.

[25] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, "A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each," pp. 3–13, 2012.

[26] C. S. Xia, Y. Wei, and L. Zhang, "Automated program repair in the era of large pre-trained language models," pp. 1482–1494, 2023.

[27] Y. Li, D. Choi, J. Chung, N. Kushman, J. Schrittwieser, R. Leblond, T. Eccles, J. Keeling, F. Gimeno, A. Dal Lago *et al.*, "Competition-level code generation with AlphaCode," *Science*, vol. 378, no. 6624, pp. 1092–1097, 2022.

[28] S. Peng, E. Kalliamvakou, P. Cihon, and M. Demirer, "The impact of AI on developer productivity: Evidence from GitHub Copilot," *arXiv preprint arXiv:2302.06590*, 2023.

[29] A. Ziegler, E. Kalliamvakou, X. A. Li, A. Rice, D. Rifkin, S. Simister, G. Sittampalam, and E. Aftandilian, "Productivity assessment of neural code completion," in *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming (MAPS)*. ACM, 2022, pp. 21–29.

[30] H. Pearce, B. Ahmad, B. Tan, B. Dolan-Gavitt, and R. Karri, "Asleep at the keyboard? assessing the security of GitHub Copilot's code contributions," in *Proceedings of the 43rd IEEE Symposium on Security and Privacy (SP)*. IEEE, 2022, pp. 754–768.

[31] R. Khoury, A. R. Avci, J. Brunelle, and B. Marc Camara, "How secure is code generated by ChatGPT?" 2023.

[32] J. W. Lin, E. K. Jones, D. J. Jasper, E. J.-s. Ho, A. Wu, A. T. Yang, N. Perry, A. Zou, M. Fredrikson, J. Z. Kolter *et al.*, "Comparing ai agents to cybersecurity professionals in real-world penetration testing," *arXiv preprint arXiv:2512.09882*, 2025.

[33] M. Shahin, M. A. Babar, and L. Zhu, "Continuous integration, delivery and deployment: A systematic review on approaches, tools, challenges and practices," *IEEE Access*, vol. 5, pp. 3909–3943, 2017.

[34] S. Kadavath, T. Conerly, A. Askell, T. Henighan, D. Drain, E. Perez, N. Schiefer, Z. Dodds, N. DasSarma, E. Tran-Johnson, S. Johnston, S. El-Showk, A. Jones, N. Elhage, T. Hume, A. Chen, Y. Bai, S. Bowman, S. Fort, D. Ganguli, D. Hernandez, J. Jacobson, J. Kernion, S. Kravec, L. Lovitt, K. Ndousse, C. Olsson, S. Ringer, D. Amodei, T. B. Brown, J. Clark, N. Joseph, B. Mann, S. McCandlish, C. Olah, and J. Kaplan, "Language models (mostly) know what they know," *arXiv preprint arXiv:2207.05221*, 2022.

[35] A. Yildiz, S. G. Teo, Y. Lou, Y. Feng, C. Wang, and D. M. Divakaran, "Benchmarking llms and llm-based agents in practical vulnerability detection for code repositories," *arXiv preprint arXiv:2503.03586*, 2025.

[36] Berkeley Risk and Decisions Initiative, "Frontier ai's impact on the cybersecurity landscape (paper summary and blog)," https://rdi.berkeley.edu/frontier-ai-impact-on-cybersecurity/, 2025.

[37] K. Greshake, S. Abdelnabi, S. Mishra, C. Endres, T. Holz, and M. Fritz, "Not what you've signed up for: Compromising real-world LLM-integrated applications with indirect prompt injection," *arXiv preprint arXiv:2302.12173*, 2023.