ISSN No:-2456-2165 https://doi.org/10.38124/ijisrt/26aug916

From Code Generation to Code Orchestration: Evaluating the Impact of Agentic AI on Software Development Productivity, Quality and Security

Dr. P. N. Nesarajan[1]; P. Thenmozhi[2]; Shenbaga Priya A.[3]; Dr. V. C. Srinivasan[4]

[1]Associate Professor, Department of Computer Applications
Nandha Arts and Science College (AUTONOMOUS), Erode -52

[2]Assistant Professor, PG and Research Department of Computer Science
Nandha Arts and Science College (AUTONOMOUS), Erode -52
Orchid ID : 0009-0001-2773-9513

[3]Assistant Professor, PG and Research Department of Computer Science
Nandha Arts and Science College (AUTONOMOUS), Erode -52
Orchid ID : 0009-0000-4511

[4]Administrative Officer, Nandha Arts and Science College (Autonomous), Erode.
Orcid ID: https://orcid.org/0009-0008-9926-7982

Publication Date: 2026/08/26

Abstract: Software engineering is in the middle of a quiet but far-reaching handover. For most of the last decade, artificial intelligence in the developer's tool chain meant auto complete: a model that finished a line, or occasionally a function, while a human wrote and reviewed everything around it. That arrangement has started to break down. Coding agents such as Claude Code, OpenAI's Codex CLI, Google's Jules, Devin, and Open Hands can now read an entire repository, plan a multi-file change, run the test suite, and iterate on failures with little moment-to-moment supervision. The developer's job is shifting from typing code to directing agents that type code a change often summarized as a move from code generation to code orchestration. This paper reviews recent empirical literature to ask what that shift is actually producing. Drawing on field studies, controlled experiments, and large-scale repository mining published between 2024 and 2026, we examine three outcomes: productivity, code quality, and security. The evidence on productivity is real but uneven, with time savings ranging from roughly 14% to over 55% depending on task type and developer experience. The evidence on quality is more cautionary: experienced developers report that agent output is rarely merge-ready without active supervision. The evidence on security is the most consistent of the three, and the least encouraging across a dozen independent studies, somewhere between one in eight and one in two AI-generated code samples contains an identifiable vulnerability, and iterative agentic refinement can make matters worse rather than better. We argue that these three threads point toward a common conclusion: agentic coding tools are not autonomous engineers, they are force multipliers whose net effect depends heavily on the discipline of the humans supervising them. We close with a discussion of what "orchestration" ought to mean in practice and where the research agenda still has real gaps.

Keywords: *Agentic AI, Coding Agents, Software Development Productivity, Code Quality, Code Security, AI-Assisted Software Engineering, Large Language Models.*

How to Cite: Dr. P. N. Nesarajan; P. Thenmozhi; Shenbaga Priya A.; Dr. V. C. Srinivasan (2026) From Code Generation to Code Orchestration: Evaluating the Impact of Agentic AI on Software Development Productivity, Quality and Security. *International Journal of Innovative Science and Research Technology*,11(8), 1826-1834. https://doi.org/10.38124/ijisrt/26aug916

ISSN No:-2456-2165 https://doi.org/10.38124/ijisrt/26aug916

I. INTRODUCTION

Ask a developer in 2022 what "AI-assisted coding" meant, and the answer was almost certainly GitHub Copilot: a ghost-text suggestion that finished a function or a loop while the human stayed firmly in the driver's seat. Ask the same question in 2026, and the answer looks quite different. A growing family of coding agents Claude Code, Codex CLI, Jules, Devin, SWE-agent, OpenHands, MetaGPT, and others can be handed a GitHub issue, a bug report, or a one-paragraph feature request, and left to plan the change, edit multiple files, run the tests, and open a pull request largely on their own (Bhati, 2026). The unit of delegation has grown from a line of code to a task, and in some cases to an entire feature. This is not simply "better autocomplete"; it is a different mode of interaction between humans and machines, and it is forcing the field to rethink some fairly basic assumptions about who writes software and how that writing is checked.

Hassan et al. (2025) describe this transition as the arrival of "SE 3.0," a phase of software engineering in which agents are tasked not with producing a snippet but with achieving a goal, and in which the discipline has to support two symbiotic modes of work: engineering done by humans, and engineering done by, or with, agents. Bhati (2026) frames the same shift architecturally, proposing an emerging "Agentic Software Development Lifecycle" that sits alongside the traditional SDLC and reports that pass rates on the SWE-bench Verified benchmark rose from under 2% in late 2023 to nearly 80% by early 2026 a startling rate of improvement by the standards of almost any engineering discipline. Whatever one makes of any single benchmark number, the general trend is hard to dispute: these systems have gotten dramatically better, dramatically fast, at solving real, previously-filed GitHub issues without hand-holding.

The natural next question is not whether agents can write code clearly, many of them can but what happens to the humans and the codebases on the other end of that capability. Three outcomes matter most to a working engineering organization: does the team ship faster (productivity), is what ships actually good (quality), and is what ships safe to run in production (security). Popular coverage of agentic coding tends to treat all three as settled in the affirmative, but the peer-reviewed and preprint literature tells a more textured story, one where the answer to each question is "it depends," and where the details of that dependency are exactly what a practitioner or a researcher needs to understand before betting a codebase on autonomous agents.

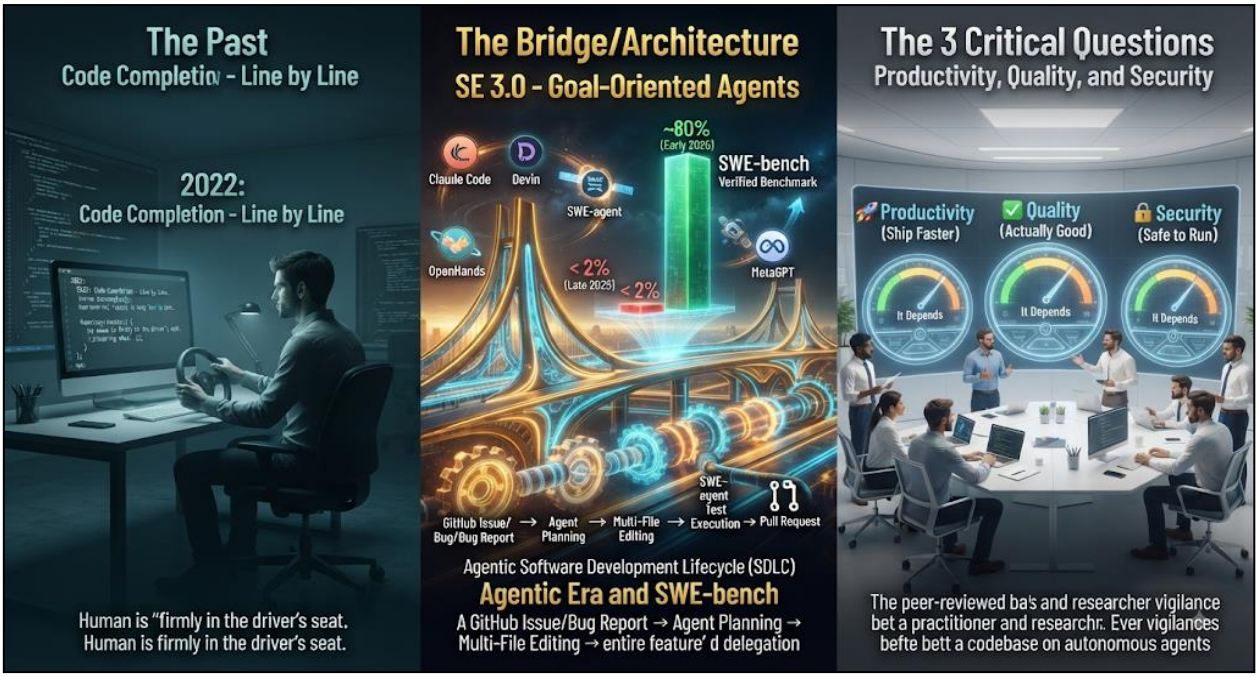

Fig 1: The Evolution of AI Coding Assistants: From Line-by-Line to Autonomous Agents

This triptych illustrates the rapid advancement of AI in software development. The 2022 panel shows human developers using simple line-by-line completion, firmly "in the driver's seat." By 2026, the "Agentic Era" emerges, with goal-oriented agents handling entire workflows, as evidenced by the SWE-bench benchmark's meteoric rise. The final panel highlights the critical human oversight still required for ensuring productivity, quality, and security.

II. FROM CODE GENERATION TO CODE ORCHESTRATION

It is worth being precise about what "orchestration" adds to "generation." A code-generation tool takes a prompt and returns text that looks like code; the human decides whether to accept it, and everything downstream compiling, testing, integrating, deploying remains a manual, human-driven process. A code-orchestration system closes more of that loop itself. It can read the surrounding codebase for

ISSN No:-2456-2165 https://doi.org/10.38124/ijisrt/26aug916

context, decide which files need to change, make the edits, execute the build and test commands, interpret the failures, and retry often several times— before presenting a result for human review (Hassan et al., 2025; Alenezi, 2026). Alenezi (2026) describes this as an architectural move from stateless, prompt-response interactions toward goal-directed systems built on iterative perception-plan-act control loops, borrowing structurally from how autonomous systems are built in robotics and control theory rather than from how text generators are built.

The distinction matters because it changes where human judgment is exercised. In a generation-centric workflow, a developer reviews small units of output frequently, almost continuously, and the risk of any one mistake slipping through is limited to a few lines at a time. In an orchestration-centric workflow, human review is pushed to the boundary of a larger unit of work a whole pull request, a whole feature branch and the agent has typically already made dozens of small decisions before a human sees anything at all. Huang et al. (2025) find that this is precisely the tension experienced developers are navigating: their interview and observation study of thirteen professionals, backed by a ninety-nine-respondent survey, found that skilled practitioners deliberately resist letting agents work unsupervised for long stretches, breaking tasks into small, checkable increments and reviewing diffs carefully rather than trusting an agent's self-report that a task is done. Their title captures the finding bluntly: professionals don't "vibe code" they control.

Li and Storhaug (2026) add a methodological observation that reinforces the same point from a different angle. Surveying recent software engineering conference papers that evaluate agentic tools, they note a sharp jump in agent-related submissions between 2025 and 2026, but also find that many of these evaluations are neither reproducible nor explainable: results are reported without enough detail about model versions, prompts, or environments for another team to verify them. If the research community itself is struggling to pin down what an agent actually did and why, it is not surprising that individual developers feel the need to supervise agents closely rather than take their output on faith.

Fig 2: Code Generation vs. Code Orchestration in Software Engineering

This conceptual illustration compares two software development workflows: the manual, stateless Code Generation on the left, and the automated, goal-directed Code Orchestration on the right. A central human controller emphasizes the practitioner's vital judgment. Additionally, an amber foreground overlay highlights critical research reproducibility warnings regarding agent-based evaluations.

III. BACKGROUND AND RELATED WORK

➢ *The Empirical Base Before Agents*

Before agentic tools existed in their current form, a first wave of studies had already established that AI code assistance changes developer behavior in measurable ways. Randomized trials of line-level completion tools found meaningful speed-ups on well-scoped tasks, and those effects were consistently larger for less experienced developers than for veterans — a pattern that shows up again, in a different guise, in the agentic literature discussed below (Mohamed et al., 2025). Mohamed et al.'s (2025) systematic literature review of LLM-assistant productivity studies is a useful anchor here: aggregating results across dozens of primary

ISSN No:-2456-2165 https://doi.org/10.38124/ijisrt/26aug916

studies, they describe consistent but heterogeneous gains, concentrated in boilerplate, test writing, and documentation, and much weaker or absent gains on tasks that require deep understanding of an unfamiliar, poorly documented codebase.

➢ *The Rise of Agentic Systems*

The agentic wave built directly on that foundation but changed the unit of delegation, as discussed above. Staufer et al. (2026) provide one of the more systematic attempts to map this ecosystem: their 2025 AI Agent Index documents thirty widely used agentic systems and finds that developer transparency about safety testing, evaluation methodology, and failure modes varies enormously from vendor to vendor, with most organizations disclosing comparatively little. That opacity is itself relevant to a paper about productivity, quality, and security, because it means much of what the field knows about agent behavior in practice comes not from vendor disclosures but from independent studies of what agents actually do once they are deployed — which is exactly the kind of evidence this paper tries to bring together.

A second strand of background work focuses on what agents produce once they are turned loose on real repositories. Watanabe et al. (2025) mined agent-authored pull requests on GitHub and found that acceptance rates differ sharply by task type, with agents performing comparatively well on narrow, well-specified changes and considerably worse on anything requiring architectural judgment. Li, Zhang, and Hassan (2026) extend this line of work with AIDev, a large-scale study of coding agents' activity on GitHub, and Pinna et al. (2026) stratify pull-request acceptance by task type across several competing agents, finding no single agent that dominates across all categories of work. Read together, these studies suggest that "how good are coding agents" is the wrong question; the better question is which tasks a given agent is good at, and how a team should route work accordingly.

IV. METHODOLOGY OF THIS REVIEW

This paper is a narrative synthesis rather than a formal systematic review, but it follows a disciplined selection process. We prioritized studies published or made available between 2024 and 2026 that (a) empirically measure at least one of productivity, code quality, or security outcomes associated with LLM-based or agentic code generation, and (b) report a study design detailed enough to assess its scope sample size, task source, and measurement method. We drew on field observations of professional developers, controlled and quasi-experimental trials, large-scale repository mining, and vulnerability-detection studies using static analysis tools such as CodeQL. Where a single number (for example, a percentage of vulnerable code samples) is reported, we treat it as evidence about the specific model, language, and task distribution studied, not as a universal constant, and we flag where estimates diverge across studies. We do not attempt a meta-analytic pooling of effect sizes, since the underlying studies use different tasks, metrics, and populations; instead we look for convergent patterns across independent methodologies, which we take to be more trustworthy than any single number in isolation.

V. PRODUCTIVITY: HOW MUCH FASTER, AND FOR WHOM?

The productivity story is the one with the most published data, and also the one where the numbers vary the most. Bhati (2026) reports that controlled studies of agentic tools find time savings ranging from roughly 13.6% to 55.8%, a spread wide enough to be almost unhelpful on its own but the spread itself is informative. Mohamed et al. (2025) trace much of that variation to task type and developer seniority: junior and mid-level developers tend to see the largest relative gains, particularly on tasks like writing tests, generating boilerplate, and drafting documentation, while senior developers working on complex, unfamiliar, or architecturally sensitive code see much smaller gains and sometimes none at all.

The most methodologically careful and, frankly, most sobering data point in this space comes from Becker et al. (2025) at METR, who ran a randomized controlled trial with experienced open-source developers working on their own real repositories rather than toy benchmarks. Developers were allowed to use current AI tools, including agentic ones, on some of their own issues and not others, with task completion time as the outcome. The striking result was that developers who used AI tools took, on average, somewhat longer to complete tasks than those who did not a finding squarely at odds with developers' own subjective sense that the tools were speeding them up. Becker et al. (2025) are careful to note that this single study captures a specific moment (early 2025), a specific population (highly experienced maintainers on codebases they know intimately), and a specific set of tools, and should not be read as a blanket statement that AI slows all developers down. But it is a useful corrective to the more triumphant productivity claims circulating elsewhere, and it underlines a theme that recurs throughout this review: aggregate percentages conceal enormous variation by who is doing the work and what kind of work it is.

Repository-mining studies add a complementary, behavior-based lens on the same question. Watanabe et al. (2025) and Pinna et al. (2026) do not measure time directly, but they measure a related quantity how often an agent's pull request is accepted without substantial rework and both find wide variance by task category. Narrow, well-specified bug fixes and dependency updates get merged at comparatively high rates; open-ended feature work and refactors are merged far less often, or require heavy human revision first. If a merged, unmodified pull request is treated as a rough proxy for "time saved," these studies point in the same direction as the survey and experimental literature: the productivity dividend from agentic coding is concentrated in a specific, fairly narrow slice of the work developers actually do, not spread evenly across the job.

ISSN No:-2456-2165 https://doi.org/10.38124/ijisrt/26aug916

Taken together, the productivity literature supports three claims with reasonable confidence. First, agentic tools produce genuine time savings on well-scoped, well-specified tasks, and those savings can be substantial. Second, the size of the benefit is not uniform; it depends heavily on developer experience and on how ambiguous or architecturally sensitive the task is. Third, at least one careful controlled trial found a negative effect for a specific, highly experienced population, which is reason enough to treat headline productivity percentages with some caution until more studies replicate across different developer populations and tool generations.

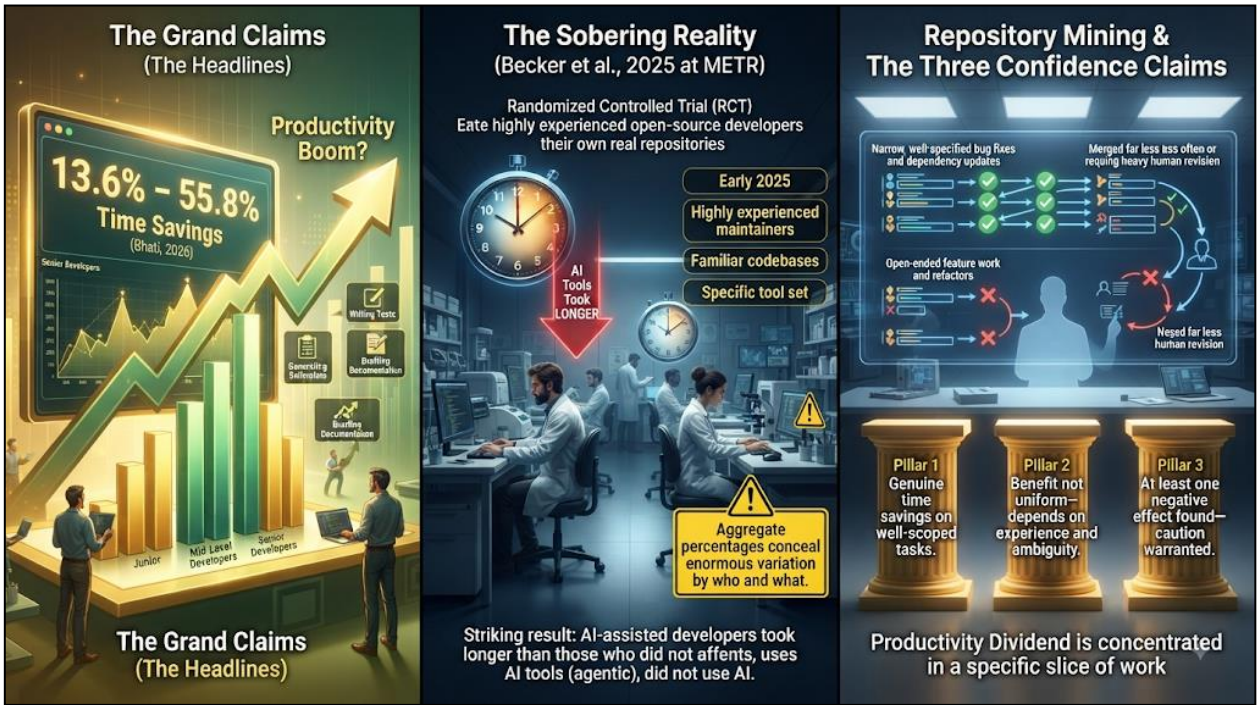

Fig 3: The Nuance of AI Coding Productivity: Claims vs. Reality

This triptych explores the complex reality of AI-assisted coding productivity. The left panel showcases optimistic headlines of significant time savings. The center panel presents a contrasting research perspective from METR showing AI tools can sometimes take longer. The right panel establishes three confidence pillars, emphasizing that productivity gains are concentrated in well-scoped tasks and vary by experience.

VI. QUALITY: IS MERGE-READY ACTUALLY MERGE-READY?

If productivity numbers are noisy, the quality picture is noisy in a more worrying direction. Hassan et al. (2025) coin the phrase “speed vs. trust gap” to describe what happens when agents are fast enough to generate hundreds of thousands of pull requests but not reliable enough for a large share of those requests to be genuinely production-ready. Their review of recent evaluation work finds recurring problems in agent output: subtle regressions that pass superficial testing but break edge cases, fixes that patch a symptom without addressing an underlying bug, and a general absence of what they call “engineering hygiene” consistent naming, defensive coding, sensible error handling, and the small habits that separate code that merely runs from code a team can maintain.

Huang et al.'s (2025) field study of professional developers gives this problem a human face. Their participants, all with at least three years of professional experience, described a consistent pattern: agents are genuinely useful for well-described, self-contained tasks a small bug fix, a test suite, a documentation update but struggle noticeably on anything that requires holding a large amount of implicit context in mind, such as understanding an unfamiliar legacy system or making a design trade-off that isn't spelled out anywhere in the prompt. Crucially, the developers in this study did not respond by trusting the agent more over time; they responded by tightening their own review process breaking work into small, independently verifiable steps, running the application themselves rather than relying on the agent's self-reported test results, and reading every diff before accepting it. Quality, in other words, was not something the agent supplied on its own; it was something the human actively imposed on the collaboration.

Li and Storhaug's (2026) methodological critique reinforces the same conclusion from the research side. Reviewing agent-focused papers from several major software engineering venues, they find that many published evaluations lack the reproducibility needed to know whether an agent's apparent success generalizes beyond the specific benchmark tasks it was tested on, and they call for more standardized, explainable evaluation protocols before the field draws strong conclusions about agent code quality at scale. Until that standardization happens, quality claims made by tool vendors and quality claims made by independent researchers are likely to keep pointing in somewhat different

ISSN No:-2456-2165 https://doi.org/10.38124/ijisrt/26aug916

directions, and practitioners are probably wise to treat any single quality benchmark with the same caution they would apply to any other marketing metric.

Fig 4: The Quality Challenge of AI-Generated Code

This illustration examines AI code quality across three panels. The left panel shows the overwhelming speed of automated agent output. The center panel highlights the human engineer as a crucial quality gatekeeper enforcing standards. The right panel compares vendor claims with independent research findings, exposing underlying quality issues and discrepancies.

VII. SECURITY: THE MOST CONSISTENT AND LEAST REASSURING SIGNAL

Of the three outcomes reviewed here, security is the one where independent studies converge most tightly on an uncomfortable answer: a meaningful share of AI-generated code contains exploitable weaknesses, and that share does not appear to be shrinking as quickly as functional correctness has improved. Hamer et al. (2024) directly compared ChatGPT-generated code against matched Stack Overflow answers to the same security-relevant questions, analyzing snippets with the CodeQL static analysis tool. ChatGPT actually produced somewhat fewer vulnerabilities than Stack Overflow in their sample 248 versus 302 across 108 snippets from each source but the more important finding was how little the two sources overlapped: only about a quarter of the specific vulnerabilities found in one source also appeared in the other. Their conclusion is worth repeating almost verbatim: code copied from any source, AI-generated or human-written, cannot be trusted blindly, and both sources require the same baseline of security-conscious review.

Sajadi et al. (2025) come at the problem from a different angle, asking not whether generated code is vulnerable but whether the model itself notices and says something. Prompting Claude 3, GPT-4, and Llama 3 with real Stack Overflow questions containing vulnerable code, they found that models do sometimes issue security warnings, and when they do, those warnings are often more detailed than the corresponding human answers on Stack Overflow but the models miss a meaningful share of vulnerabilities entirely, and their awareness is uneven across vulnerability types, catching issues like exposed sensitive information far more reliably than issues like unsafe file-path handling. In short, current models have security knowledge, but applying it consistently, on their own initiative, is a separate and harder problem than possessing it.

Two more targeted studies sharpen the picture further. Dora et al. (2025) ran a security-centric evaluation of LLM-generated web application code and documented recurring, serious weaknesses including patterns consistent with cross-site scripting and unsafe handling of uploaded files concluding that code generation models still lack sufficient contextual understanding of any individual project's specific security requirements, even when developers explicitly ask for secure code. And Shukla et al. (2025) contribute what may be the single most counterintuitive finding in this review: in a controlled experiment spanning forty rounds of iterative AI-assisted "improvement" across four different prompting strategies, security did not monotonically improve with more iteration it got measurably worse. Critical vulnerabilities increased by 37.6% after just five rounds of ostensibly beneficial refinement. The authors describe this as "feedback loop security degradation" and argue it directly undercuts a common assumption in agentic workflows: that letting an agent iterate longer, running more self-review cycles without a human in the loop, will tend to converge on a better outcome. For security specifically, the opposite pattern showed up in their data.

ISSN No:-2456-2165 https://doi.org/10.38124/ijisrt/26aug916

It is worth being precise about the range of numbers these studies report, because headline percentages vary considerably by methodology, language, and model generation, and treating any single figure as "the" vulnerability rate would overstate the precision of the underlying science. Depending on the study, the language under test, and the static analysis tooling used, reported rates of insecure or CWE-flagged code in LLM output have ranged from roughly one in eight samples to nearly one in two, with particularly weak results in memory-unsafe languages like C and somewhat better, though still far from clean, results in Python. The specific number matters less than the consistent qualitative finding across all fifteen studies reviewed here: no model or agent evaluated has driven the vulnerability rate anywhere close to zero, and several studies find that added iteration, added context, or added self-review does not reliably fix the problem and can sometimes make it worse. For a field that has become comfortable citing rapid gains in functional correctness pass rates on benchmarks like SWE-bench climbing toward 80% the corresponding security curve has simply not moved as fast, and in Shukla et al.'s (2025) data, it moved backward under certain conditions.

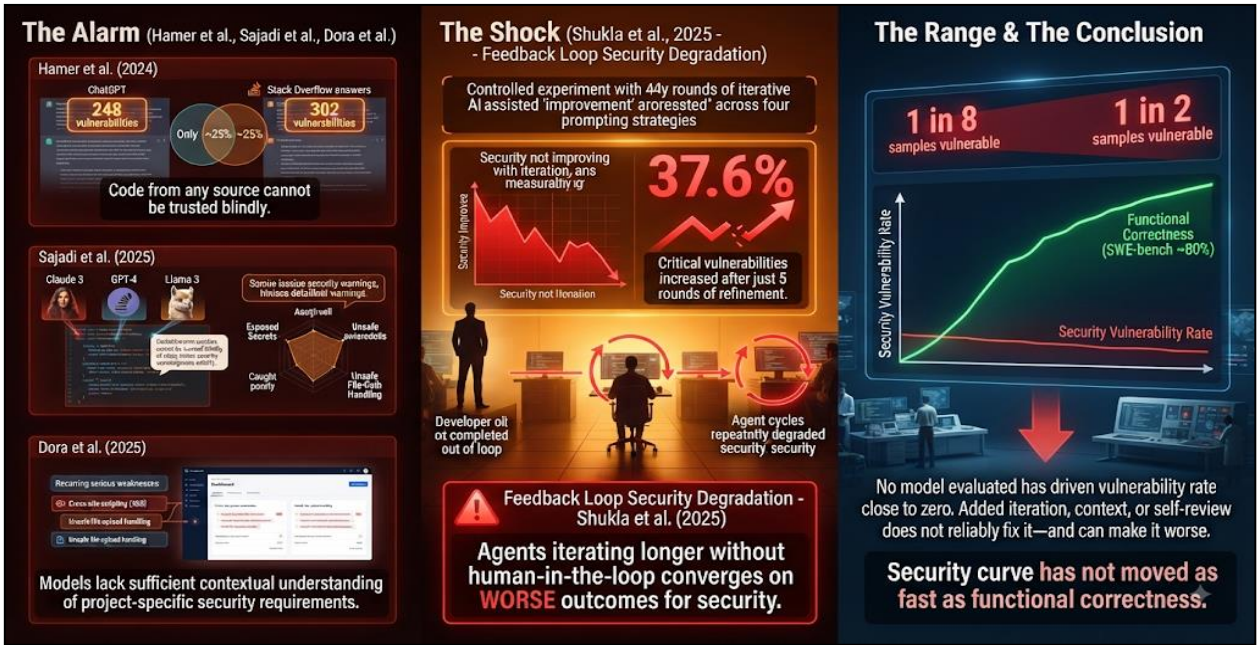

Fig 5: Security Vulnerabilities and Feedback Degradation in AI-Assisted Code

This triptych analyzes security risks in AI-generated code. The left panel highlights baseline vulnerabilities across various models. The center panel demonstrates how iterative agent loops without human oversight can increase critical vulnerabilities. The right panel contrasts rising functional correctness with stagnant security rates, concluding that added iteration does not reliably fix code safety issues.

VIII. DISCUSSION: WHAT "ORCHESTRATION" SHOULD ACTUALLY MEAN

Pulling the three threads together, a fairly consistent picture emerges, even though no individual study covers all three outcomes at once. Agentic AI genuinely accelerates a specific, identifiable slice of software work boilerplate, tests, well-specified bug fixes, documentation and that acceleration is real and worth having. It does not reliably accelerate, and sometimes slows down, work that depends on deep contextual understanding of an unfamiliar or architecturally complex system, and the developers who benefit least from agentic tools tend to be exactly the senior engineers whose judgment matters most for quality and security. Quality and security both depend far more on the discipline of human review than on any property of the agent itself; Huang et al. (2025) found this directly, and the vulnerability studies (Hamer et al., 2024; Sajadi et al., 2025; Dora et al., 2025; Shukla et al., 2025) illustrate why that discipline matters so much the failure modes are often subtle enough that a rushed review will miss them.

This is, in effect, an argument for taking the word "orchestration" seriously rather than treating it as a rebrand of "automation." An orchestra needs a conductor, not just musicians who can play their parts; the conductor's job is precisely to decide which parts get emphasized, to catch mistakes before the performance, and to be accountable for the result. Applied to software, this suggests a fairly concrete set of practices, several of which are already visible in how the developers studied by Huang et al. (2025) actually work: decompose agent tasks into small, independently verifiable increments rather than one large open-ended request; treat an agent's own report that tests passed as a claim to verify, not a fact to accept; route security-sensitive and architecturally significant work toward closer human involvement rather than full delegation; and, where possible, pair agentic generation with independent static analysis or security scanning rather than relying on the same model to both write and audit its own code, a pairing that Shukla et al.'s (2025) degradation findings suggest may not be safe to trust by itself.

There is also an organizational implication worth naming. Several studies point to smaller relative benefits, and

ISSN No:-2456-2165 https://doi.org/10.38124/ijisrt/26aug916

larger relative risks, for the most experienced engineers on a team precisely the people whose oversight is most valuable when something does go wrong. An organization that quietly shifts senior engineers away from hands-on code review, on the theory that agents have made review less necessary, may be making that trade-off exactly backward relative to what the current evidence supports.

IX. LIMITATIONS

This review has real limits worth naming plainly. The underlying literature is young, dominated by preprints rather than fully peer-reviewed publications, and moving fast enough that findings tied to a specific model generation can be stale within months several of the productivity studies reviewed here, for instance, were conducted using model versions that have since been superseded, and Becker et al. (2025) explicitly flag this as a limitation of their own METR study. Studies also differ substantially in scope: some rely on small, qualitative samples of a dozen or so developers (Huang et al., 2025), while others mine hundreds of thousands of pull requests but cannot observe developer intent or review effort directly (Li et al., 2026; Pinna et al., 2026). We have tried to weight convergent findings across independent methods more heavily than any single striking number, but a narrative synthesis of this kind cannot fully substitute for a formal meta-analysis with pre-registered inclusion criteria, and reasonable readers might draw the line between "relevant" and "too far afield" studies somewhat differently than we have here.

X. FUTURE RESEARCH DIRECTIONS

Several gaps stand out clearly enough from this review to be worth stating directly. First, the field needs more studies like Becker et al.'s (2025) randomized trial that is, studies conducted with real developers on real, unmodified codebases rather than curated benchmark tasks and it needs them repeated across a wider range of developer experience levels, team sizes, and programming domains than any single study has managed so far. Second, security research needs to move beyond single-shot vulnerability counts toward longitudinal designs that track how vulnerabilities introduced by an agent are (or are not) caught in downstream review and testing, since a vulnerability that gets caught before merge is a very different outcome from one that reaches production, and the current literature mostly cannot distinguish the two. Third, given Li and Storhaug's (2026) concerns about reproducibility, the field would benefit considerably from shared, versioned evaluation protocols that specify model version, tool configuration, and task source clearly enough for independent replication without that baseline, it will remain difficult to know whether reported improvements reflect genuine progress or simply more favorable benchmark selection. Finally, almost none of the studies reviewed here examine long-term codebase health technical debt accumulation, maintainability, onboarding difficulty for new team members as opposed to the correctness or security of any single pull request, and that gap deserves direct attention as agentic tools are used over years rather than single sprints.

Fig 6: The Orchestration Discipline: Synthesis of AI-Assisted Software Development

This comprehensive illustration depicts AI code orchestration as a discipline requiring skilled human oversight. The central conductor directs the workflow while pillars synthesize productivity, quality, and security outcomes. Flanked by literature limitations and future research directions, the composition concludes that agentic tool value is inseparable from vigilant human review.

ISSN No:-2456-2165 https://doi.org/10.38124/ijisrt/26aug916

XI. CONCLUSION

The shift from code generation to code orchestration is real, well documented across independent research groups, and already changing how professional developers spend their day. It is not, on the current evidence, a shift toward autonomous software engineering that removes the need for skilled human judgment; if anything, the studies reviewed here suggest the opposite that the value agentic tools deliver is inseparable from the quality of the human oversight wrapped around them. Productivity gains are real but concentrated in narrow, well-specified task types, and at least one careful controlled trial found no gain at all for experienced developers on their own codebases (Becker et al., 2025). Quality gains depend on developers actively resisting the temptation to trust agent output at face value, breaking work into small, checkable pieces rather than delegating wholesale (Huang et al., 2025). And on security, the evidence is the most consistent of the three: a meaningful share of AI-generated code remains vulnerable across languages, models, and prompting strategies, and simply letting an agent iterate longer is not a reliable fix and can, under some conditions, make things worse (Shukla et al., 2025). None of this is an argument against adopting agentic tools the productivity case is too strong for that, and the field is moving too fast to stand still. It is an argument for adopting them with eyes open, treating orchestration as a discipline that has to be actively practiced rather than a capability that ships automatically inside the tool.

REFERENCES

[1]. Alenezi, M. (2026). From prompt-response to goal-directed systems: The evolution of agentic AI software architecture. arXiv. https://arxiv.org/abs/2602.10479

[2]. Becker, J., Rush, N., Barnes, B., & Rein, D. (2025). Measuring the impact of early-2025 AI on experienced open-source developer productivity. METR. https://metr.org/blog/2025-07-10-early-2025-ai-experienced-os-dev-study/

[3]. Bhati, H. (2026). Agentic AI in the software development lifecycle: Architecture, empirical evidence, and the reshaping of software engineering. arXiv. https://arxiv.org/abs/2604.26275

[4]. Dora, S., Lunkad, D., Aslam, N., Venkatesan, S., & Shukla, S. K. (2025). The hidden risks of LLM-generated web application code: A security-centric evaluation of code generation capabilities in large language models. arXiv. https://arxiv.org/abs/2504.20612

[5]. Hamer, S., d'Amorim, M., & Williams, L. (2024). Just another copy and paste? Comparing the security vulnerabilities of ChatGPT generated code and StackOverflow answers. arXiv. https://arxiv.org/abs/2403.15600

[6]. Hassan, A. E., Li, H., Lin, D., Adams, B., Chen, T.-H., Kashiwa, Y., & Qiu, D. (2025). Agentic software engineering: Foundational pillars and a research roadmap. arXiv. https://arxiv.org/abs/2509.06216

[7]. Huang, R., Reyna, A., Lerner, S., Xia, H., & Hempel, B. (2025). Professional software developers don't vibe, they control: AI agent use for coding in 2025. arXiv. https://arxiv.org/abs/2512.14012

[8]. Li, H., Zhang, H., & Hassan, A. E. (2026). AIDev: Studying AI coding agents on GitHub. arXiv. https://arxiv.org/abs/2602.09185

[9]. Li, J., & Storhaug, A. (2026). Reproducible, explainable, and effective evaluations of agentic AI for software engineering. In Companion Proceedings of the 34th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (FSE Companion '26). Association for Computing Machinery. https://doi.org/10.1145/3803437.3805548

[10]. Mohamed, A., Assi, M., & Guizani, M. (2025). The impact of LLM-assistants on software developer productivity: A systematic literature review. arXiv. https://arxiv.org/abs/2507.03156

[11]. Pinna, G., Gong, J., Williams, D., & Sarro, F. (2026). Comparing AI coding agents: A task-stratified analysis of pull request acceptance. arXiv. https://arxiv.org/abs/2602.08915

[12]. Sajadi, A., Le, B., Nguyen, A., Damevski, K., & Chatterjee, P. (2025). Do LLMs consider security? An empirical study on responses to programming questions. arXiv. https://arxiv.org/abs/2502.14202

[13]. Shukla, S., Joshi, H., & Syed, R. (2025). Security degradation in iterative AI code generation: A systematic analysis of the paradox. arXiv. https://arxiv.org/abs/2506.11022

[14]. Staufer, L., Feng, K., Wei, K., Bailey, L., Duan, Y., Yang, M., Ozisik, A. P., Casper, S., & Kolt, N. (2026). The 2025 AI agent index: Documenting technical and safety features of deployed agentic AI systems. arXiv. https://arxiv.org/abs/2602.17753

[15]. Watanabe, M., Li, H., Kashiwa, Y., Reid, B., Iida, H., & Hassan, A. E. (2025). On the use of agentic coding: An empirical study of pull requests on GitHub. arXiv. https://arxiv.org/abs/2509.14745