AIDA: Asymmetric Impact of AI Assistance in Software Development Teams

Ekolabs.ai

2025

Abstract

In late 2025, a mid-sized startup software team implemented an AI coding assistant (Anthropic's Claude Code) during a six-week pilot study. The core insight from this implementation is that the AI did not simply boost the volume of output; it fundamentally transformed the nature of the team's work. Routine coding tasks were completed approximately 20% faster, enabling additional time to be allocated to more complex tasks (with a 27% increase in time spent) and to enhancing code quality. Key quality metrics improved: average pull request (PR) size decreased by about 47%, indicating more focused and incremental changes, and the proportion of code commits containing tests rose from 26% to 47%, a 21 percentage-point increase. The team did not deliver more features in less time; rather, they produced higher-quality output within the same timeframe by completing simpler tasks more quickly, addressing complex problems more thoroughly, and substantially increasing testing. These findings highlight the asymmetric impact of AI assistance, with tangible benefits distributed unevenly across work types and team members. We introduce the AIDA (Asymmetric Impact of Developer AI-Assistance) framework as a way to make these distributional changes visible and discuss implications for CTOs and researchers.

1 Study Context

The study involved a team of four developers at a mid-sized startup, comprising three software engineers (Dev1, Dev2, Dev3) and one QA engineer. The team's workflow, including agile sprints and code reviews, and the codebase remained consistent throughout the observation period. A baseline "PRE" period (three sprints prior to the introduction of Claude Code) and a "POST" period (three sprints following adoption, after a brief transition) were established.

During the POST period, developers utilized the Claude Code assistant for tasks such as code writing, test generation, and solution research, and received coaching on effective tool usage. Metrics collected included development speed, code quality, and work patterns. Traditional productivity measures (e.g., cycle time, velocity) were supplemented with distributional metrics to assess AI's impact on various task types. For instance, the study tracked completion times for the easiest (P25) and most difficult (P75) tasks, as well as code quality indicators such as bugs per task, percentage of work involving testing, and average pull request size.

These metrics were analyzed using the AIDA framework, which emphasizes both capacity freed (time saved on routine work) and the subsequent utilization of that capacity (e.g., on complex tasks or quality improvements). In practice, the metrics can be aligned with existing ceremonies (such as sprint reviews, retrospectives, and technical backlog reviews) to provide deeper insight into how work shifts once AI tools are introduced.

2 Key Results: The "Freed Capacity" Pattern

At the team level, the introduction of the AI tool produced a distinct pattern. Routine tasks, which form a large share of daily coding work, were completed about 20% faster. At the same time, more complex and demanding tasks received about 27% more time and deeper engagement. In other words, the team reallocated effort: time saved on easy tasks was reinvested in difficult ones and in quality improvements.

This reallocation helps explain why traditional productivity metrics, such as average task completion time or total output, might initially appear unchanged. When measured only by throughput, the team looks similar before and after AI adoption. However, the composition of work changed substantially: simpler work accelerated, more complex work was tackled with greater depth, and quality-related activities (especially testing) increased. The productivity gain is asymmetric: it appears in where and how effort is applied, not only in how much is delivered.

3 Individual Outcomes and Asymmetry

The impact of the AI assistant varied significantly among team members. Despite identical conditions (same tool and training period), each developer experienced distinct outcomes.

3.1 Dev1: Speed and Quality Gains

Dev1, a senior engineer, exemplified a successful case. Active coding time per task decreased by approximately 65% (excluding time spent on non-coding activities), with improvements across all measured metrics. The proportion of commits including tests rose from 36% to 57%, a 21-point increase. This substantial increase in test-related commits indicates a major gain in quality-oriented work, not just faster delivery.

Average PR size for Dev1 decreased by about 38%, indicating more incremental changes that are typically easier to review and safer to deploy. Importantly, Dev1's bug rate remained stable, suggesting that faster coding did not come at the cost of more defects. This constitutes a near-ideal scenario for AI adoption: substantial speed gains alongside maintained or improved quality.

3.2 Dev2: The "Capacity Liberated" Effect

Dev2, another experienced developer, exhibited a different yet positive pattern that we describe as the "capacity liberated" effect. Dev2's average time per task doubled post-AI adoption (+101% time per story point), which could initially be misinterpreted as a productivity decline. However, a closer percentile analysis clarified this outcome.

Easy tasks (P25) were completed 44% faster, while difficult tasks (P75) took 47% longer than before. In practice, Dev2 accelerated routine work with AI assistance and utilized the freed time to address more challenging problems that might previously have been postponed or completed hastily. Supporting this interpretation, Dev2's pull requests became 61% smaller on average, and test-writing activity nearly doubled (test commit ratio increased from 10.6% to 20.0%).

Rather than delivering a higher quantity of items, Dev2 achieved greater improvements by dedicating additional time to complex tasks and increasing test coverage. The bug rate per unit of work remained roughly constant, indicating that quality was maintained despite the increased difficulty of the work taken on. This illustrates how AI assistance can enable developers to accelerate routine tasks and expand into more complex, higher-quality work—a positive outcome that is not captured well by aggregate velocity metrics alone.

3.3 Dev3: Friction and Testing-Focused Gains

Dev3, by contrast, experienced mixed results. Having previously used AI coding tools such as Cursor and OpenAI's Codex, Dev3 did not exhibit a dramatic change in coding throughput following the introduction of Claude. Post-adoption data indicated that Dev3's coding tasks took longer on average, with active coding time per point increasing by 109%. Even the P25 easy tasks took 40% longer, suggesting friction or inefficiencies in adapting to the new tool.

It is plausible that Dev3 encountered challenges with AI-generated suggestions or spent time iterating with the assistant without net benefit, a scenario reported by some developers when AI tools do not align with their coding style or the structure of the codebase. Suspected blockers include tool–workflow mismatch, lack of trust in AI-generated outputs, and ingrained prior habits. Naming these friction sources can help teams diagnose similar cases.

Nevertheless, Dev3 derived value from the AI in other areas. Analysis and research tasks were completed 57% faster, and testing contributions increased substantially. Prior to adoption, only about 2.5% of Dev3's commits included tests; post-adoption, this figure rose to nearly 48%. The test file touch ratio increased from nearly 0% to 19.4%, indicating a significant shift toward testing.

Quality also improved slightly, with an approximate 2% reduction in bugs per story point. In Dev3's case, the AI assistant was leveraged primarily for testing and exploratory programming rather than accelerating feature development. While the output of new code did not increase—and may have slowed due to integration overhead or caution—the net effect on test coverage and design exploration was positive. This case highlights that productivity gains from AI assistance may not be uniform; some developers may benefit primarily in secondary tasks such as testing or research, especially during early stages of adoption.

3.4 QA Engineer: New Capabilities Rather Than Speed

The QA engineer achieved a notable outcome. Unlike developers, the QA role did not involve story-point tasks, precluding measurement of metrics such as hours per point. Instead, qualitative and indirect impacts were assessed.

During the trial, the QA engineer received additional AI coaching and leveraged the assistant to develop new test automation that had previously been considered infeasible within existing time constraints. This effort resulted in the introduction of 504 new performance tests to the codebase, where none had existed before. The QA engineer reported high satisfaction with the AI tool and indicated that it significantly empowered their work.

To give a sense of potential strategic value, we can perform a simple illustrative calculation: if we assume that historically such a volume of additional tests might prevent on the order of 50 bugs, and if each bug is estimated to cost around $300 in time and incident resolution, then the tests could avert approximately $15,000 in defect-related costs over their lifespan. This back-of-the-envelope estimate is intentionally approximate, but it helps translate quality improvements into a budgetary perspective that is more legible to executives.

The QA engineer's use of AI also had spillover effects. Team-wide improvements in testing—such as higher overall test coverage and a greater frequency of test commits—coincided with this period, even though those effects cannot be attributed to a single individual metric. The QA case is a clear example of AI creating qualitatively new capabilities rather than simply making existing tasks faster.

3.5 Asymmetric Impact Within One Team

The diverse experiences of Dev1, Dev2, Dev3, and the QA engineer exemplify the asymmetric impact of AI assistance. Even within a single team and project, the tool amplified individual strengths and preferences in distinct ways. Two developers (Dev1 and Dev2) demonstrated clear gains in efficiency or capability, while one (Dev3) experienced a more complex outcome, with improvements in certain areas and setbacks in others.

This aligns with reports from larger-scale studies: not all developers benefit equally from AI coding assistants. Microsoft and GitHub's experiment on GitHub Copilot found an average 55.8% speed increase on coding tasks, but the gains were heterogeneous; developers with less experience or those coding more hours per day tended to benefit most, while others saw more modest improvements.[2] In our case, the variation did not simply follow seniority, as Dev3 was not a novice, but rather reflected the influence of personal workflow, prior AI experience, and task selection on individual outcomes.

4 Quality, Productivity, and the "AI Productivity Paradox"

A notable aspect of this study is the importance of how productivity is defined. If only traditional productivity metrics are considered—such as the number of tasks completed or the speed of feature delivery—the team's performance would appear largely unchanged or only slightly improved after adopting AI. For example, one developer's official cycle time (Dev1's overall time from task start to completion) improved by approximately 7.8%, despite a 65% increase in active coding speed, and Dev2's average throughput per point decreased on paper due to increased time invested in each task.

A superficial assessment based on these metrics alone could lead management to conclude that the AI tool was ineffective or not valuable. However, this would be a misleading conclusion. A more comprehensive analysis revealed significant value creation in code quality, test coverage, and the resolution of complex work—benefits not captured by traditional metrics. This phenomenon, in which individual productivity gains do not translate into obvious team output gains, has been observed elsewhere and is sometimes described as an "AI productivity paradox". Syntheses of GitHub research, Stack Overflow surveys, and industry reports show teams experiencing rapid coding output with AI but mixed results in overall delivery and satisfaction.[3]

Evidence from GitHub's own studies suggests that AI-assisted code can surpass human-only code in quality: in recent experiments, code written with Copilot support scored higher on functionality, readability, and maintainability in blind reviews.[5] In our case, the team produced more tests and smaller, more manageable code changes, which are indicative of higher quality and reduced risk in software engineering.

Freeing developers from repetitive tasks appears to yield benefits beyond code metrics alone. Surveys indicate that a large majority of developers using AI assistants report higher job satisfaction and reduced frustration, as they can focus on more rewarding work rather than boilerplate coding.[6] Although our study did not formally measure job satisfaction, qualitative feedback suggested that developers valued the opportunity to allocate time to more challenging or interesting problems (e.g., Dev2 engaging more deeply with complex tasks, or the QA engineer creating new test suites). The enthusiasm around quality-focused work is consistent with broader findings that AI tools can help preserve mental energy for creative problem-solving by automating routine work.

At the same time, these findings and external studies caution that AI assistance is not a universal solution and can introduce new inefficiencies if not used appropriately. The case of Dev3, where some tasks took longer with AI, illustrates that these tools can sometimes slow down experienced

users when suggestions are inaccurate or when additional time is spent managing AI-generated output. Many developers report that they do not fully trust AI suggestions and spend extra time reviewing or correcting AI-generated code.[3] If an AI assistant suggests incorrect or suboptimal code, developers may need to debug or rewrite it, potentially negating any time saved.

In our study, developers received coaching on best practices, such as verifying suggestions and favoring AI for boilerplate and test generation, but validation overhead can still diminish benefits. AI amplifies productivity when its guidance is accurate, but it can hinder progress when it is not. Developing the skill to discern when and how to rely on AI is therefore essential. In Dev3's case, overuse or misuse of AI for certain coding tasks may have reduced efficiency, while significant benefits were realized in test writing, a domain where AI suggestions are often more reliable. This suggests a strategic approach: align the AI tool's strengths with appropriate tasks. Routine code and testing are clear areas of strength, as noted by other engineering teams.[4]

5 Implications for CTOs and Engineering Teams

For technology leaders, particularly CTOs at mid-size startups, these results have several implications for the adoption of AI developer tools.

First, success criteria for AI assistance should extend beyond raw output. If a CTO tracks only metrics such as features shipped per quarter or story points completed, immediate improvements may not be apparent, potentially leading to the mistaken conclusion that AI is not adding value. Instead, leaders should consider metrics such as:

- **Quality:** bug rates, test coverage, and code review feedback.
- **Capacity:** time saved on routine coding tasks and how that time is reinvested.
- **Complexity:** the depth and thoroughness of problem-solving and innovation.

In this case study, AI did not simply increase coding speed; it altered workflows to produce more robust code (more tests, smaller change sets) and to address deeper technical challenges. These are long-term investments in quality and maintainability that may reduce technical debt and accelerate future releases. Industry case studies report teams using Copilot achieving markedly faster lead times to production with no loss in quality, and in some cases improved code coverage.[4] Our study reflects similar directional outcomes: lead-time improvements were modest, but failure rates (bugs) did not increase despite more ambitious work, and test coverage increased substantially. The return on investment for AI tools may therefore come less from immediate feature volume and more from enabling higher-quality work and greater capacity to handle complexity.

Second, CTOs should be prepared for uneven adoption and learning periods. Not every developer will adapt to the AI tool at the same pace or with the same enthusiasm. Our study was conducted only six weeks post-adoption; some team members (such as Dev3) might require more time to integrate the tool effectively into their workflow. Research and field reports suggest giving teams a period on the order of several months to reach proficiency.[3] It is prudent to expect heterogeneity in results: some people or tasks will improve dramatically, others might stagnate or even temporarily decline. This does not necessarily indicate failure; rather, it signals where targeted support may be needed.

Personalized coaching or training can help developers who are not benefiting initially. In our case, one-on-one coaching sessions were provided, and the QA engineer—who received the most targeted coaching—used the tool very effectively. A "turn it on for everyone and hope for the

best" approach is less likely to succeed than a deliberate enablement program that includes shared examples, internal talks, and structured experimentation time.

Third, leaders must carefully consider whether to mandate the use of AI tools or to make adoption optional. If only a few team members use the tool, the organization may not realize full team-level benefits and might unintentionally create output disparities. Conversely, forcing every developer to use AI regardless of their comfort or the suitability of their tasks can backfire. Industry surveys and commentary have documented tension between executives pushing AI adoption and engineers dealing with immature tools and integration issues.[7] Many developers report frustration when AI mandates are imposed without clear success criteria, guidance, or allowances for variance in workflows.

A balanced, evidence-based approach is advisable. One strategy is to treat adoption models themselves as experiments: for example, A/B testing teams or squads with different enablement levels and gathering both quantitative metrics and sentiment data before making policy decisions. Instead of strictly forcing AI on every coding task, leaders can encourage adoption by highlighting internal success stories (e.g., Dev1's speed-and-quality gains, Dev2's capacity-liberated pattern, or the QA engineer's new tests), providing training, and allowing developers to skip AI where it demonstrably does not help.

It may even be appropriate to assign work based on AI aptitude: engineers who realize strong gains with AI might take on tasks that leverage that advantage (such as writing large volumes of tests or boilerplate-heavy features), while those less comfortable with AI can focus on tasks where human intuition and domain expertise remain central. Whether to rebalance work based on AI leverage is an open organizational question, but our results suggest that management must actively manage AI integration rather than assume uniform uplift.

6 Rethinking Metrics and the AIDA Index

A key motivation for this case study was to explore improved methods for measuring the impact of AI assistance, given the limitations of standard productivity metrics. The AIDA framework was introduced to examine distributional changes (such as P25 and P75 values) and quality metrics, rather than reducing outcomes to a single productivity figure.

The challenge is whether these changes can be meaningfully distilled into a single index or score for "AI impact". As the results show, improvements in some areas and declines in others do not average out cleanly. For example, if one developer's output speed doubles and another's remains unchanged, an average may obscure the first developer's significant success and the second's neutral experience. Similarly, if easy tasks accelerate while hard tasks deliberately slow down, a single "speed" number is not representative. A single-number metric is therefore likely to obscure important nuances.

That said, many organizations still request summary figures. One compromise is to report a *scorecard* of a small number of metrics that together characterize the asymmetric effects. For example, at the team level in our study, one can summarize:

- capacity freed on routine tasks: P25 completion times improved by approximately 20%;
- capacity reinvested on complex tasks: P75 completion times increased by approximately 27%, reflecting deeper engagement;
- quality shifts: test-commit ratio increased by 21 percentage points, and average PR size decreased by 47%.

An "AIDA scorecard" would thus consist of a short vector, such as (freed capacity, reinvested capacity, key quality deltas), rather than a single scalar.

If a single composite index is still desired, an organization could define an AI impact index as a weighted combination of normalized improvements. For example, letting:

- R denote normalized acceleration on routine work (e.g., improvement in P25 time, expressed as a positive fraction),
- Q denote a normalized quality improvement (e.g., a combined change in test coverage and PR size), and
- C denote normalized additional depth on complex work (e.g., change in P75 time, interpreted positively when increased time reflects investment rather than delay),

one could define:

$$\text{AIDA\_Index} = 0.4R + 0.4Q + 0.2C.$$

The specific weights (40/40/20) are intentionally illustrative, reflecting the intuition that routine acceleration and quality should typically carry more weight than increased time on complex tasks, but organizations can adjust these coefficients according to their priorities (for instance, assigning more weight to quality in safety-critical domains). Importantly, such an index should be accompanied by the underlying components so that decision-makers can see where improvements are actually occurring.

From an academic perspective, this challenge highlights the need for richer frameworks to assess AI-assisted productivity. The SPACE framework (Satisfaction, Performance, Activity, Communication, and Efficiency) has been proposed as a more holistic lens for evaluating developer productivity.[8] Our findings support this direction: focusing solely on performance (speed) can be misleading, whereas considering satisfaction (developer enjoyment and reduced frustration), activity (such as increased test writing), and efficiency across task types yields a more complete picture. By naming and measuring the "asymmetric impact", both practitioners and researchers are encouraged to move beyond single metrics such as lines of code or tasks per week when evaluating AI tools.

Our study has limitations. It examines a single team with a small sample size (three developers with quantitative metrics, plus one QA), over a relatively short period. The results, while internally consistent and supported by related research, should be generalized with caution. Different teams might use freed capacity differently; some might choose to ship extra features, while ours chose to improve internal quality. The effectiveness of AI assistance can also vary depending on the nature of the codebase (e.g., legacy monoliths versus newer services) and organizational constraints (such as heavy regulation).

These limitations suggest several questions for further investigation. How universal is the freed-capacity pattern? Under what conditions do teams convert freed capacity into feature throughput versus quality investments? What factors lead to a Dev3-type outcome where coding speed does not improve, and how can organizations respond? How should managers handle scenarios where some developers see 2x gains and others see little change? And how can long-term ROI be measured when near-term feature throughput does not increase but quality metrics improve? Addressing these questions will likely require longitudinal and multi-team studies, as well as open sharing of metrics and methodologies.

7 Conclusion

In summary, the introduction of AI assistance (Claude Code in this case study) did not simply accelerate code production; it changed how the development team allocated time and approached tasks. The team's experience demonstrates a nuanced productivity enhancement characterized by the asymmetric impact of AI. Efficiency gains in routine development tasks enabled the team to take on more complex, rigorous work, resulting in improved engineering outcomes such as increased testing and more granular code changes, even as traditional throughput metrics remained relatively stable.

Two of the three developers experienced clear improvements in efficiency or capability, while one exhibited a more mixed adaptation, underscoring the need for individualized support and realistic expectations during AI rollouts. For practitioners, particularly startup CTOs evaluating AI tools, it is important to recognize the value of quality and learning improvements, not just speed. With appropriate metrics and management, AI tools can contribute to a stronger product and a more empowered engineering team.

For the academic and research community, this case emphasizes the necessity of multi-dimensional approaches to measuring AI's impact. A narrow focus on coding speed risks overlooking substantive changes in work patterns and value creation. Expanding the definition of productivity to include both capacity freed and its effective reuse is therefore recommended. The AIDA framework represents an initial step in this direction, with further research needed to refine it into a robust methodology.

As AI assistance becomes increasingly prevalent in software development, organizations that deliberately leverage its asymmetric impact—accelerating routine tasks while enabling greater creativity and complexity—are likely to derive the greatest benefit. The objective is not merely to do the same work faster, but to enable developers to allocate their time more effectively, using AI as a strategic tool within the broader sociotechnical system of the team.

References

[1] Ekolabs.ai. *AIDA Internal Report: Asymmetric Impact of Developer AI-Assistance*. Technical report, 2025.

[2] H. Peng, S. Ross, M. Muller, et al. *The Impact of AI on Developer Productivity: Evidence from GitHub Copilot*. arXiv:2302.06590, 2023.

[3] V. Siedykh. *AI Development Team Productivity: Analysis of GitHub Research and Community Studies*. groCTO Substack, 2025.

[4] Faros AI. *Is GitHub Copilot Worth It? Real-World Data Reveals the Answer*. Faros AI Engineering Blog, 2023.

[5] J. Bauer. *Does GitHub Copilot Improve Code Quality? Here's What the Data Says*. GitHub Blog, 2024.

[6] I. Shani and GitHub Staff. *Survey Reveals AI's Impact on the Developer Experience*. GitHub Blog, 2023.

[7] S. Lazzaro. *AI Coding Mandates Are Driving Developers to the Brink*. LeadDev, 2025.

[8] N. Forsgren, M. Storey, and C. Maddila. *The SPACE of Developer Productivity: There's More to It Than You Think*. Communications of the ACM, 64(10), 2021.