Writing Code vs. Shipping Code: Productivity Effects Across Generations of AI Coding Tools*

Mert Demirer† Leon Musolff‡ Liyuan Yang§

September 6, 2026

Abstract

How do the productivity effects of AI evolve across successive generations of tools, and to what extent do task-level gains ultimately translate into final output? We study these questions in the context of software development, using data on more than 500,000 GitHub developers combined with their AI usage telemetry. In a matched event study design, we find that autocomplete, interactive coding agents, and autonomous coding agents each significantly increase coding activity ("commits"), with respective cumulative effects of 30%, 180%, and 240%. These gains, however, attenuate sharply across the production hierarchy: the 240% cumulative effect falls to 80% for the number of projects, and to 30% for actual releases. This pattern is consistent with the weak-link hypothesis: the strong productivity gains from AI are attenuated by human bottlenecks in the production chain, with an estimated elasticity of substitution of 0.23 between AI and human effort, which indicates strong complementarities. We further confirm these results across four major software marketplaces, finding a sharp increase in the number of new apps but no increase in total usage. Large task-level AI productivity gains have therefore translated only partially into shipped and used software thus far.

JEL: O33, D24, J24, L86, L17

Keywords: generative AI, productivity, software development, production function

*We thank Joel Becker, Seth Benzell, Alexander Bick, Adam Blandin, David P. Byrne, Tom Cunningham, Andrey Fradkin, Jan De Loecker, Anders Humlum, Sanjog Misra, Frank Nagle, Sida Peng, Nate Rush, Suproteem Sarkar, Max Schnidman, Chad Syverson and Kevin Xu. This paper benefited from comments by seminar participants at BIG.AI@MIT, the Chicago AI Workshop, the RAP Symposium at Harvard & MIT, the PSU AI and the Economy Initiative, the St. Louis Fed, the Government Office of the Czech Republic, Wharton's AI and the Future of Work Conference, the Columbia MAD conference, HKU, NUS and HKUST. This project was funded in part by the Mack Institute For Innovation Management at Wharton and the Chicago AI Incubator. Frederick Lei and Roman Shrestha provided excellent research assistance.

†MIT, mdemirer@mit.edu

‡Wharton, lmusolff@wharton.upenn.edu

§BU, liyuany@bu.edu

1 Introduction

Generative AI differs from earlier technologies in how fast it has improved and diffused: model capabilities have advanced rapidly, and firms deploy AI across a growing range of applications (Bick et al., 2025; Kalyani et al., 2025; Humlum and Vestergaard, 2025; Yotzov et al., 2026). While much of the existing literature emphasizes the productivity effects of individual AI tools on specific tasks, far less is known about two questions that are central to understanding the economic impact of AI. How do productivity effects evolve across generations of tools? And how much of the task-level gains reaches final output?

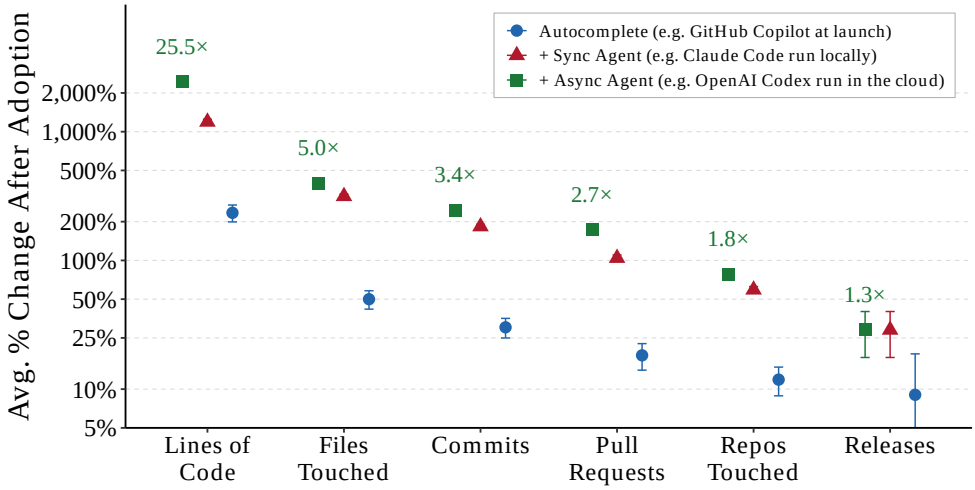
We study these questions in software development, one of the earliest and most prominent domains of AI adoption (Handa et al., 2025) and among the occupations most exposed to AI (Felten et al., 2023; Eloundou et al., 2024; Tomlinson et al., 2025). Software development provides an ideal setting for two reasons. First, it has seen three generations of widely adopted tools: *autocomplete*, which suggests code as the developer types; *sync agents*, which write and edit code alongside the developer in real time; and *async agents*, which can be assigned a task and work autonomously until it is complete. Second, its production process is a well-defined sequence of stages—writing code, integrating changes across files, reviewing and merging pull requests, managing releases—so we can measure productivity at each point along the chain.

Our first contribution is to the literature on how AI affects productivity in specific tasks (Noy and Zhang, 2023; Brynjolfsson et al., 2025; Cui et al., 2026; Dell'Acqua et al., 2026). We provide evidence from more than 500,000 developers on GitHub across multiple tools and four years of model improvement (2022–2026), so we track how effects evolve as capabilities advance rather than at a single point in time.

Our second contribution is to measure how much of these gains reaches final output. Task-level estimates of AI's productivity effect are large (15–50%, see the review by Fruits and Stout 2026), but several papers argue they need not translate into proportional gains in aggregate output (Acemoglu, 2025; Jones, 2026; Karger et al., 2026). The leading argument is the *bottleneck* or *weak-links* hypothesis: AI raises productivity in some tasks, but output is limited by complementary tasks that humans still perform at unchanged productivity (Kremer, 1993; Jones, 2011, 2026). We extend the weak-links framework to a hierarchical model in which software output is produced by sequentially aggregating tasks within a single production process, and provide one of the first empirical tests of the hypothesis.

In the model, output at each layer combines the previous layer's output with human input at that layer through a CES function. AI can affect a layer in two ways: by augmenting human input, so each unit of human effort produces more, or by partially automating it, so direct human effort is no longer needed but human review still is. For instance,

Figure 1: Overview of Productivity Effects of AI Coding Tools



Notes: This figure summarizes matched event-study estimates of the effect of adopting AI coding tools on lines of code, files touched, commits, pull requests, repositories touched, and releases. The underlying estimates are from Table 6. "Autocomplete" refers to AI-based code completion; "sync" to agents that work locally with the developer (e.g., Claude Code); and "async" to agents that operate autonomously until a task is completed (e.g., GitHub Agent). Because more capable tools are adopted alongside earlier generations, the figure presents cumulative effects of adopting all tools up to and including a given generation.

autocomplete augments the production of code but does not automate it; by contrast, async agents automate not only the production of code but the production of entire pull requests—which still require human review before they can be merged. The key parameter in the model is the elasticity of substitution. The lower it is, the more AI's gains are bottlenecked by the human stages downstream and the more they attenuate along the production chain. In the case of complementarities, even unbounded automation yields only finite gains, as in the weak-links literature (Kremer, 1993; Jones, 2011).

To take the model to the data, we combine public GitHub data with internal Microsoft telemetry to measure activity at each layer of the hierarchy and to date each developer's adoption of AI tools. Combining publicly observable AI usage on GitHub with internal Microsoft data lets us track multiple tools across vendors: autocomplete from GitHub Copilot; sync agents from GitHub Copilot, Claude, and Codex; and async agents from GitHub Copilot and Codex. To measure the productivity effects of each tool, we implement an event-study design in which each treated user is matched with a similarly active control user in the same calendar week one year earlier. Drawing controls from the prior year avoids contamination from private AI use among contemporaneous non-adopters.

Figure 1 summarizes our results, plotting the cumulative effect of each generation of tools on six outcomes spanning the production hierarchy, from lines of code through releases. Focusing on commits, our measure of task-level output, AI coding tools raise

productivity substantially, and the effect grows with each generation. Autocomplete raises weekly commits by about 30%; adding sync agents brings the cumulative effect to about 180%; adding async agents brings it to about 240%. These effects are larger for less active developers, though they remain substantial throughout the activity distribution.

How do these task-level gains translate into final output? Effects attenuate sharply up the production hierarchy. Developers write much more code and work on more features after adoption, but ship only modestly more software. Sync agents raise lines of code by 958% and pull requests by 86%, but releases by only 20%. Autocomplete shows the same pattern: 234% on lines of code, 30% on commits, 9% on releases. Aggregate GitHub activity corroborates the developer-level estimates: new repositories and pull requests have risen sharply since early 2025, in a pattern consistent with the attenuation we estimate.

We next examine two mechanisms that could generate this attenuation: an increase in *scope* and *experimentation*. Under increased scope, the productivity gains do reach the higher layers, but they accrue through an increase in the size of each unit of output rather than in the number of units produced. Under experimentation, developers generate more coding activity at the earlier stages, where AI is most effective, but most of it does not reach final output. This is the natural implication of the complementarities in our model: code review, quality control, and release capacity limit how much of the additional early-stage output advances to the later stages, where human input remains essential.

We find that increased scope accounts for only a small fraction of the attenuation and that most of it reflects experimentation. The evidence is visible at every layer of the hierarchy. After adopting sync agents, developers add 7.0 times as many lines of code but delete 12.2 times as many, so more of what they write is rewritten or discarded. They create 79% more branches, exploring features that are often later abandoned. And they try more project ideas that go nowhere: the share of new repositories receiving no further activity after their first month has risen 9 percentage points since 2021, from 60% to 69%.

Having shown evidence consistent with complementarities, we next quantify them directly by mapping our estimates to the model's parameters. The estimated attenuation pattern is consistent with an elasticity of substitution of 0.23 between upstream output and human effort, indicating strong complementarity between AI and human inputs. The estimates are also consistent with autocomplete entering the production hierarchy only at the code-writing layer, while sync and async agents intervene at later stages as well. We also investigate whether the empirical results are consistent with developers reallocating their inputs across layers in response to AI, and find no evidence of full reoptimization. This is consistent with the slow adjustment of production processes to new general-purpose technologies (Bresnahan and Trajtenberg, 1995; Brynjolfsson et al., 2021).

Our evidence so far shows an increase in developer-side output, which does not tell us whether new apps reach end users or how widely they are used. To address these questions, we assemble monthly panels for four of the largest software marketplaces: the Apple App Store, the Google Play Store, the Chrome Web Store, and SourceForge. For each marketplace, we observe the first date that each app appeared and a measure of cumulative end-user engagement (the number of ratings or downloads). With these data, we quantify whether the number of new apps has increased over time and, if so, whether this translates into greater usage.

New apps have risen across all marketplaces since early 2025, with monthly entry roughly doubling on average. Usage, however, has not followed: total engagement with new apps after launch has not increased in any marketplace. While this rules out market expansion effects of AI, one possibility is that these apps still raise consumer surplus through better matching—consumers use or switch to niche apps that are better matched to their tastes—even though total usage remains the same. Our evidence does not support this hypothesis either: the share of new apps that fail to reach even a modest audience has risen everywhere, so the expansion is concentrated in apps with little or no user base. This pattern admits two interpretations: either the marginal apps are of low quality, or there is an additional bottleneck on the consumer side—discovery and adoption of new apps take time—that prevents supply-side gains from translating into usage. Our data currently cannot distinguish between the two.

We use several strategies to address the endogeneity concerns inherent in observational data and to rule out alternative explanations for the attenuation pattern we document. First, to mitigate activity bias, we require control users to be similarly active on the treated user's adoption date and match on pre-treatment activity over the prior eleven weeks. Second, we validate our results with placebo tests that apply the same event-study design to non-AI coding tools and find no effects. Third, we document the absence of differential pre-trends between treated and control users across all outcomes, supporting the parallel trends assumption. Fourth, we show that our results are robust to several choices in matching and empirical specification. Fifth, we compare our autocomplete estimates with the prior field-experimental evidence of Cui et al. (2026), who estimate the effect of the same tool in the same time period. Sixth, we show that the productivity effects and the attenuation are not driven by changes in what we measure: neither a decline in the quality of commits nor an increase in the verbosity of AI-generated code accounts for them.

Three caveats are worth noting about the external validity of our findings. First, our data cover only open-source software and four major software marketplaces. While open-source software is a large segment of the market and a key input into proprietary software (Nagle,

2019; Gortmaker, 2026), our data have little coverage of enterprise software. However, other research shows that the same attenuation pattern holds in enterprise software (Chen and Stratton, 2026), with effects considerably smaller than ours. This is expected, as open-source developers work in small teams and face less organizational friction, so realizing productivity gains from AI requires less complementary investment (Brynjolfsson et al., 2021). In this sense, we think our setting is a leading indicator of how AI will affect the broader industry. Second, generative AI has advanced furthest in software engineering relative to other domains. While this lets us test general principles about the impact of AI, our quantitative estimates may not extrapolate to other domains. Third, although our data are nearly contemporaneous, extending through mid-2026, our estimates necessarily reflect the capabilities of the tools available to date. How much these bottlenecks loosen going forward will depend on whether future tools need less human review or automate higher levels of the hierarchy, and on how quickly they spread across stages of production.

Relation to Literature Our work relates to four strands of literature. First, a growing literature studies the effects of generative AI tools on productivity across settings, including coding (Peng et al., 2023; Kreitmeir and Raschky, 2024; Chen et al., 2025; Becker et al., 2025; Song et al., 2025; Cui et al., 2026; Sarkar, 2026; He et al., 2026; Chen and Stratton, 2026), customer support (Brynjolfsson et al., 2025), law (Choi and Schwarcz, 2025), consulting (Dell'Acqua et al., 2026), writing (Noy and Zhang, 2023), entrepreneurship (Otis et al., 2026), hiring (Jabarian and Henkel, 2026), and AI use in startups (Kim et al., 2026). With some exceptions, these papers find productivity effects on the order of 15–50% or more (Fruits and Stout, 2026). Other papers show that AI changes not only productivity but also how workers allocate time across tasks: away from coordination and communication, and toward individual production work (Hoffmann et al., 2024; Dillon et al., 2025). A related literature studies aggregate labor-market outcomes and finds modest employment effects to date (Hampole et al., 2025; Humlum and Vestergaard, 2025), though Brynjolfsson et al. (2026) document declines for early-career workers in the most AI-exposed occupations.[1]

Most of these estimates are for early-generation tools—autocomplete and chatbots—while developers are shifting to agentic tools (Johnston et al., 2026). Systematic evidence on agentic tools remains limited and mixed: Becker et al. (2025) find that AI coding tools slow experienced developers down, while other studies report substantial productivity gains (Chen et al., 2025; Sarkar, 2026). Closest to our paper, Chen and Stratton (2026) find that AI tools raise coding activity in enterprises, but that these gains only partially pass

[1]Relatedly, Autor and Thompson (2025) and Freund and Mann (2026) study how automation reshapes occupations by changing the tasks that remain—through the expertise required by remaining tasks in the former, and through worker sorting across occupations by comparative advantage in the latter.

through to output. The papers differ in four ways. They study firms; we study open-source development. Their evidence on the bottleneck is direct, coming from longer review times, whereas we quantify the strength of complementarities by developing and calibrating a model of software production. They examine labor market effects, whereas we examine the demand side, asking whether the additional software reaches users. Finally, we emphasize how effects change across three generations of tools as the technology improves.

Second, our conceptual framework builds on the task-based approach to automation (Autor et al., 2003; Acemoglu and Autor, 2011; Acemoglu and Restrepo, 2018, 2019, 2022), in which AI displaces human labor in some tasks but not others. In this literature, Acemoglu (2025) considers how task-level productivity gains aggregate to economy-wide output, and Demirer et al. (2026), Gans and Goldfarb (2026), Garicano et al. (2026), and Agrawal et al. (2026) consider how automation—or augmentation—propagates through interdependencies and chains of tasks.[2] When tasks are complementary, automating a subset of them has bounded effects on aggregate output—the "weak links" or "O-ring" logic of Kremer (1993) and Jones (2011), which Aghion et al. (2019) and Jones (2026) extend to AI. We apply this logic vertically within a single production process: the "tasks" are the layers of the software production hierarchy, and the weak link is the human-bottlenecked layer with the smallest AI-driven productivity gain. This yields a specific, testable prediction—that productivity gains attenuate through the hierarchy—which we confirm empirically.[3]

Third, a growing literature studies how generative AI reshapes output and consumption on digital platforms. Goldberg and Lam (2026) document a large supply expansion alongside crowd-out of incumbents on a stock-images marketplace; Reimers and Waldfogel (2026) find that new book releases on Amazon tripled between 2022 and 2025, while average quality declined; and Zhou and Lee (2024) find that AI adoption raises creators' output and engagement on an image-sharing platform. We extend this work to software, documenting an analogous supply expansion across major software marketplaces.

Finally, our aggregate analysis connects to a longstanding debate about whether and how quickly productivity gains at the micro level translate into aggregate output growth, and about the complementary organizational investments required for them to do so (Solow, 1987; Bresnahan et al., 2002; Bloom et al., 2012; Brynjolfsson et al., 2021). We provide early evidence on this question for AI in the context of software development.

[2]Relatedly, Restrepo (2025), Korinek and Suh (2024), and Davidson et al. (2026) study economies approaching full automation, asking whether growth or wages remain constrained by what is not automated.

[3]Our model is also related to hierarchical models of knowledge work, in particular Garicano (2000) and Ide and Talamàs (2025), for which Kim and Koning (2026) document evidence that AI-native startups have flatter hierarchies than comparable firms. While the hierarchies in these models are in skill—less knowledgeable workers handle routine tasks while more knowledgeable workers handle exceptions—we consider a hierarchy in the production process, aggregating more granular outputs into higher-level ones.

2 Background

This section outlines a typical developer workflow and the generations of generative AI coding tools we study.

2.1 Developer Workflow

Software developers write code in integrated development environments (IDEs), such as Visual Studio Code (VS Code), which provide syntax highlighting, error checking, and debugging tools. Code is organized in plain-text source *files*, grouped into projects and modules. Projects are managed with version-control systems, most commonly Git. Developers periodically bundle a set of related changes, describe them briefly, and record the result as a *commit*, the fundamental, reversible unit of version control.

Multi-developer projects are coordinated through collaboration platforms such as GitHub. One core feature of these platforms is an issue tracker (e.g., GitHub Issues), where developers describe tasks, discuss requirements, and link to relevant code. Once they agree on a task, a developer is assigned to the issue and implements the changes.

To work in parallel, developers create separate *branches* of the codebase, each of which can be edited independently without immediately incorporating others' changes. Once work on an issue is finished, the developer submits a *pull request*, which indicates that the changes are ready for review; other developers inspect the code, request modifications, and ultimately merge it into the project's main branch. Each project lives in its own *repository* (a project-specific folder), and a developer may contribute to several repositories at once. Finally, for easier distribution, a repository can be packaged into *releases*—compiled code associated with a version number and a changelog.

2.2 A Brief History of Generative AI Tools in Software Development

Software development emerged as an early application domain for generative AI due to the inherently structured nature of computer code and the availability of large training data. Since 2022, a wide range of generative AI coding tools have become available, evolving rapidly in scope and autonomy. We distinguish three categories: autocomplete, synchronous (sync) agents, and asynchronous (async) agents.

Autocomplete GitHub launched GitHub Copilot in June 2022, before ChatGPT, as the first widely adopted generative AI coding assistant.[4] Copilot runs inside the IDE and completes code as the developer types, generating snippets, comments, and documentation

[4]While earlier IDE-integrated code-completion tools existed, GitHub Copilot was the first to be widely adopted; many others have since followed with similar offerings, such as Codeium (late 2022), Cursor (March 2023), and Amazon CodeWhisperer (April 2023).

from the surrounding context.[5]

Sync Agents Beginning in early 2025, several AI-assisted coding tools evolved to support an agentic framework in which a developer submits a prompt describing a task, and the agent works on it directly in the IDE or from the command line. Unlike chatbot-based workflows, in which developers typically copy and paste suggested code, the agent can navigate the codebase, propose and apply edits across multiple files, execute code (e.g., running unit tests), and iteratively refine its actions in real time. The developer watches the agent work, accepts or rejects its changes, and is still the one who commits them. Since the developer supervises the agent as it works, we call these sync agents.

Async Agents A more recent development is async coding agents, which operate autonomously for extended periods with little or no human interaction. Developers assign a high-level task—a feature, a bug fix, a refactor—and the agent plans and executes it, so async agents can take on larger and more complex tasks than sync agents.

In May 2025, GitHub introduced the *GitHub Agent*, which operates directly within GitHub. The agent is invoked by assigning it an issue, in the same way as assigning a task to a human collaborator; see Appendix Figure OA-3(a) for an example. Upon assignment, the agent launches on a cloud-based virtual machine, autonomously identifies relevant code, makes edits, runs tests, and pushes commits to a draft pull request. Once the task is completed, the developer is notified and can review the proposed changes, request revisions, or merge the pull request.

OpenAI Codex offers similar async functionality, autonomously modifying code and opening pull requests.[6] Codex differs from the GitHub Agent in how tasks are assigned: rather than through a GitHub issue, developers assign tasks through an external interface on OpenAI's website. Appendix Figure OA-3(b) shows an example task assignment.

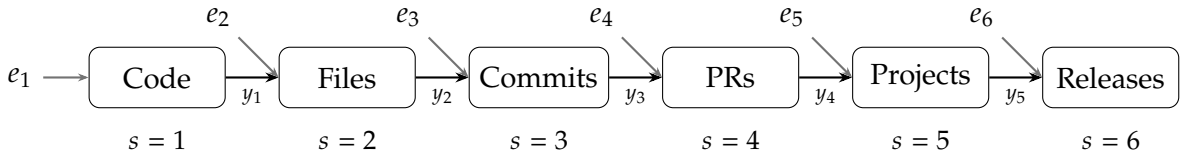
While our classification of AI coding tools into these generations follows the evolution of the tools, the line between sync and async agents has blurred more recently: most sync tools today (such as Claude Code) can also operate in async mode, executing tasks autonomously for extended periods.[7] The practical distinction thus rests on the complexity

[5]Following the launch of ChatGPT in November 2022, developers also quickly adopted AI chatbots for coding-related tasks. In September 2023, GitHub Copilot and Cursor integrated chat-based functionality directly into IDEs, reducing context switching and giving the chatbot direct access to the codebase. Because chatbot usage spans many platforms and is not coding-specific, we do not analyze chatbots as a separate category.

[6]Several other async agents can be initiated directly on GitHub, including Cursor Agents, Google Jules, and Devin, but these collectively account for less than 10% of async agent activity on public repositories and are excluded from our analysis. We also exclude "vibe coding" tools such as Replit and Lovable, which are primarily targeted at non-developers.

[7]Claude Code added an async mode in October 2025. Meanwhile, OpenAI released a Codex desktop app supporting synchronous use in February 2026.

Figure 2: A Representation of the Software Production Function



Notes: This figure depicts the software production hierarchy used throughout the paper. Each layer s combines the output of the layer below, y_{s-1}, with the effective input devoted to that layer, e_s: lines of code combine into files, files into commits, commits into pull requests, pull requests into projects, and projects into releases. At the first layer, output is produced from effective input alone ($y_1 = e_1$). Effective input at each layer combines the developer's own effort with AI-generated input, as defined in Equation (3), and final output is $Y \equiv y_6$.

of the assigned task and the degree of developer supervision rather than on the tool itself. As a result, some of the sync-agent effect we estimate may partly reflect asynchronous use by developers we classify as sync adopters. This affects only the attribution of effects between the sync and async modes, not the total effect of agentic tools.

3 A Model of Software Production

The previous section described how software production proceeds through a hierarchy of stages—from lines of code to files, commits, pull requests, and releases—and how successive generations of AI tools intervene at different points in this pipeline. We now formalize this structure in a model of software production. All proofs are given in Appendix A.

3.1 Setup

Production proceeds through S layers, ordered from most granular to most aggregate. At the lowest layer, the developer writes lines of code. At each subsequent layer $s \geq 2$, output of layer $s-1$ is converted into output of layer s using additional input, e_s. This process is illustrated in Figure 2, where lines of code are combined into files, files into commits, and commits into pull requests. In this process, the additional input is the effort of review, coordination, and design: checking that a file is correct, deciding which changes belong together in a commit, and integrating commits into a coherent pull request.

Production at each layer. At the first layer, where lines of code are produced, output is given by:

$$y_1 = e_1, \tag{1}$$

where e_1 denotes the effective input devoted to writing code, which may come from the human, the AI, or both, and is formally defined below. At each subsequent layer $s \geq 2$,

production combines upstream input with effective input, e_s, via a CES technology:

$$y_s = \left[\alpha_s \, y_{s-1}^{\frac{\sigma_s - 1}{\sigma_s}} + (1 - \alpha_s) \, e_s^{\frac{\sigma_s - 1}{\sigma_s}}\right]^{\frac{\sigma_s}{\sigma_s - 1}}, \qquad s = 2, \ldots, S, \tag{2}$$

where y_{s-1} is the output of layer $s-1$, e_s is the effective input at layer s, $\alpha_s \in (0,1)$ is the share parameter, which governs the relative importance of y_{s-1} and e_s in production, and $\sigma_s > 0$ is the elasticity of substitution between upstream output y_{s-1} and effective input. Final output is $Y \equiv y_S$.

Effective input. We now define the effective input e_s, which combines the developer's own effort with input generated directly by AI:

$$e_s = B_s h_s^p + \min(a_s, \; \phi_s h_s^r) \tag{3}$$

where $h_s = h_s^p + h_s^r$ is the developer's own effort at layer s, split between time spent generating output directly (h_s^p) and time spent reviewing AI-generated input (h_s^r), and $a_s \geq 0$ is the quantity of AI-generated input at layer s. Each input has a cost—developer time for h_s^p and h_s^r and inference and prompting costs for a_s.

AI enters effective input in two ways. The first is through $B_s \geq 1$, which multiplies the developer's own effort and therefore captures a direct productivity effect. We normalize $B_s = 1$ in the absence of AI, so that one unit of developer effort produces one unit of effective input at layer s. The second is through the Leontief term $\min(a_s, \; \phi_s h_s^r)$, where AI directly creates input at layer s, but the developer must spend review time h_s^r for that output to become productive. The parameter $\phi_s \geq 0$ therefore indexes the quality of AI input—the higher is ϕ_s, the less review time the developer must spend for each unit of AI input to become productive. These two parameters, B_s and ϕ_s, capture distinct channels through which AI affects production:

Definition 1 (Augmentation)**.** *Augmentation arises when $B_s > 1$: the developer's own effort is amplified through the term $B_s h_s^p$. AI makes the human more productive at layer s—for example, by suggesting code completions or highlighting potential issues during code review.*

Definition 2 (Partial automation)**.** *Partial automation arises when $\phi_s > 0$: AI-generated output at layer s becomes productive with some human review. Automation is only partial because human input remains essential: without human review, AI input has no value.*

Definition 3 (Full automation)**.** *In the limit $\phi_s \to \infty$, an arbitrarily small amount of review suffices: for any $h_s^r > 0$, $\min(a_s, \phi_s h_s^r) \to a_s$. Automation is full because the human is no longer needed at layer s.*

3.2 Discussion

Augmentation vs. automation. In the model, augmentation and automation do not coexist within a layer. Because $B_s h_s^p$ and $\min(a_s, \phi_s h_s^r)$ enter e_s as perfect substitutes in Equation (3), the developer's optimal allocation at each layer is a corner solution: she devotes her effort either to direct production—augmented by B_s—or to reviewing AI-generated input (a_s). This is by design: contemporary AI coding tools are typically used in one mode at a time, not blended within a layer. Therefore, the two channels capture economically distinct use cases of AI, and h_s^r makes explicit the review and integration effort that defines how developer work has changed.[8]

Generality. Our analysis focuses on the specific layers of software production, but the framework applies more generally. For example, some layers may correspond to creative and design work—for instance, architectural decisions about how to structure a codebase, or the initial idea for a feature. These layers are analogous to the "idea production function" in the growth literature (Aghion et al., 2019; Jones and Tonetti, 2026), and the same logic applies: AI can generate candidate designs or feature proposals ($a_s > 0$), but human judgment is required to evaluate and select among them (ϕ_s finite).

Relationship to the production function literature. While we set up the model hierarchically, the resulting technology is a nested CES production function (see Appendix A.1 for the derivation). Nested CES is a natural representation for hierarchical production because it is weakly separable across nests, with each nest homothetic: the inputs of a nest can be summarized by a single index that enters the nest above (Shephard, 1953; Demirer, 2026). In our setting, this intermediate input is the output of the previous layer, y_{s-1}, which is then combined with effective input e_s at layer s.

3.3 Where AI Coding Tools Enter the Production Hierarchy

We now describe how the three generations of AI coding tools we study map to the model's parameters—specifically, which layers they augment ($B_s > 1$), which layers they automate ($a_s > 0$), and which layers remain purely human. Table 1 summarizes the mapping.

Autocomplete. Autocomplete tools help the developer write code faster in the first layer by proposing code snippets as they type:

$$e_1 = B_1 h_1^p, \qquad B_1 > 1, \qquad \phi_1 = 0. \tag{4}$$

[8] If using AI were costless, the augmentation parameter B_s and the automation parameter ϕ_s would play identical roles—each simply multiplies a unit of human effort to deliver effective input ($B_s h_s^p$ versus $\phi_s h_s^r$). What distinguishes them is the cost of using AI in practice, which has two components: a direct monetary cost (inference) and a time cost (prompting, configuration, and review).

Table 1: Where AI Coding Tools Enter the Production Hierarchy

Tool	$s = 1$ (Code)	$s = 2$ (Files)	$s = 3$ (Commits)	$s = 4$ (PRs)	$s = 5$ (Projects)	$s = 6$ (Releases)
Autocomplete	Augmenting	—	—	—	—	—
Sync agent	Partial automation	Partial automation	Augmenting	Augmenting	—	—
Async agent	Partial/full automation	Partial/full automation	Partial/full automation	Partial/full automation	—	—

Notes: "Augmenting" denotes layers where AI raises the developer's productivity ($B_s > 1$); "partial automation" denotes layers where AI generates output subject to human review ($\phi_s > 0$); "—" indicates no AI involvement.

The developer still writes every line, but each unit of human effort h_1^p produces more output. Since autocomplete does not operate at any higher layer, all effective input above layer 1, e_s, remains purely human ($\phi_s = 0$ so $a_s = 0$ for all s, and $B_s = 1$ for $s \geq 2$). The productivity gain from autocomplete must therefore pass through every subsequent layer before reaching final output.

Sync agents. Sync agents can autonomously generate code and files, but the developer monitors the process in real time. This corresponds to AI output generation in layers 1 and 2 ($\phi_1, \phi_2 > 0$), but the developer must review and approve the output:

$$e_s = \min(a_s,\ \phi_s h_s^r), \qquad s = 1, 2. \tag{5}$$

At the higher layers, the agent does not generate commits or pull requests directly, but it can still augment the human input in these layers—for example, by summarizing changes, documenting code, or highlighting relevant context for review:

$$e_s = B_s h_s^p, \qquad B_s > 1, \qquad \phi_s = 0, \qquad s = 3, 4. \tag{6}$$

Therefore, the sync agent operates through both channels: it partially automates layers 1–2 ($\phi_s > 0$) and augments the developer at layers 3–4 ($B_s > 1$).

Async agents. Async agents operate more autonomously. Unlike sync agents, they generate not only code and files but also commits and pull requests directly, without requiring the developer's real-time involvement. The agent can generate output at all layers up to and including pull requests:

$$e_s = \min(a_s,\ \phi_s h_s^r), \qquad s = 1, 2, 3, 4. \tag{7}$$

At layers 5 and 6—projects and releases—the agent generates no output, so effective input there remains purely human ($\phi_s = 0$ and $B_s = 1$ for $s = 5, 6$).

3.4 The Elasticity of Substitution and Shock Propagation

We now explore how an AI-generated productivity gain at a given layer translates into final output, holding the allocation of human input across layers fixed for simplicity. In the long run, developers could reoptimize their input across layers in response to AI; we consider this as an extension in Section 3.5.

Consider an AI-driven productivity gain only at layer $s-1$, through either augmentation or automation, that raises output there by a factor $A > 1$ with the same human input. The pass-through to the next layer is:

$$\frac{\tilde{y}_s(A)}{y_s} = \left[\theta_s \, A^{\rho_s} + (1 - \theta_s)\right]^{1/\rho_s}, \tag{8}$$

where $\tilde{y}_s(A)$ denotes output at layer s after the productivity gain at layer $s-1$. Two parameters govern this pass-through:

$$\theta_s \equiv \frac{\partial \ln y_s}{\partial \ln y_{s-1}} = \frac{\alpha_s \, y_{s-1}^{\rho_s}}{\alpha_s \, y_{s-1}^{\rho_s} + (1 - \alpha_s) \, e_s^{\rho_s}}, \qquad \rho_s = \frac{\sigma_s - 1}{\sigma_s}. \tag{9}$$

Here θ_s is the output elasticity of y_s with respect to upstream input y_{s-1}, capturing how heavily layer s leans on layer-$(s-1)$ output rather than on effective input e_s. The second parameter, $\rho_s = (\sigma_s - 1)/\sigma_s$, reflects how substitutable y_{s-1} and e_s are at layer s. At given input levels, the pair (θ_s, σ_s) is a reparametrization of (α_s, σ_s), since Equation (9) is one-to-one in α_s. We therefore state the comparative statics in (θ_s, σ_s), the parametrization we take to the data in Section 6.3. Two results follow.

Proposition 1 (Comparative statics of pass-through)**.** *Fix a productivity shock $A > 1$ at layer $s-1$. The pass-through $\tilde{y}_s(A)/y_s$ defined in Equation (8) is:*

(a) strictly increasing in the elasticity of substitution between upstream and effective input at layer s (σ_s);

(b) strictly increasing in the output elasticity of y_s with respect to upstream input (θ_s).

This proposition shows that the key parameter determining the rate of propagation is the elasticity of substitution σ_s. When $\sigma_s < 1$ (complements), upstream output is a poor substitute for effective input. A developer who receives double the code but has the same time to review it cannot produce twice as many files, so only part of the gain passes

through. In the limit $\sigma_s \to 0$ the technology approaches Leontief, $y_s \to \min(y_{s-1}, e_s)$, and the gain vanishes completely once the unchanged human input becomes the binding factor. When $\sigma_s > 1$ (substitutes), by contrast, an increase in y_{s-1} translates more fully into y_s: because upstream output can stand in for effective input, the fixed review capacity does not hold the gain back. The proposition also shows that pass-through is increasing in θ_s: the more layer s relies on upstream output rather than its own effective input, the more of the gain it inherits.

We next present two corollaries that show how a productivity gain from AI at one layer translates into changes in output further along the hierarchy.

Corollary 1 (Attenuation and layer of entry)**.** *(i) A productivity shock at layer k attenuates as it propagates through the hierarchy: the effect on output at layer $s > k$ is strictly smaller than the effect at layer k, and the cumulative effect at layer s is the composition of pass-throughs at each intervening layer.*[9]

(ii) Consider two technologies that each raise output by the same factor $A > 1$, but one enters at layer k and the other at layer $k' > k$ (closer to final output). The effect on final output from the technology entering at layer k' is strictly larger, because the shock passes through fewer layers of attenuation.

The intuition is straightforward: at each layer, the upstream output must be combined with effective input whose productivity has not changed, so only a fraction of the gain passes through. The earlier the layer at which a shock enters, the more such bottlenecks it must traverse, and the more the gain is compressed.[10] It follows that the impact of any AI tool depends not only on its effectiveness but also on *where* it enters the production hierarchy. A tool that provides a modest improvement at a high layer may increase final output by more than a tool that provides a large improvement at a low layer. Autocomplete, for example, enters at layer 1, so its productivity gain is attenuated five times before reaching releases; agents enter as high as layer 4 and are attenuated only twice.

Proposition 2 (Shape of attenuation)**.** *The shape of pass-through depends on whether upstream and effective input are complements or substitutes:*

(a) ***Shape of attenuation across layers.*** *Assume σ and θ are common across layers, and let $G_s \equiv \tilde{y}_s / y_s$ denote the cumulative output gain at layer s following a shock A at the origin*

[9] This result requires that upstream and effective input both have positive weight at every layer ($\theta_s \in (0, 1)$).

[10] We maintain throughout that units at each layer are comparable before and after AI adoption. If AI packed more value into each unit, quantities would understate output, which would create spurious attenuation. We think this is unlikely, as generating the attenuation pattern would require value per unit to rise progressively across layers; we further test this empirically in Section 6.2.

layer. Let the survival rate $R_s \equiv \ln G_{s+1}/\ln G_s \in (0,1)$ be the share of the gain that survives one more layer. Then R_s is strictly increasing in s when $\sigma < 1$ (complements), constant at θ when $\sigma = 1$ (Cobb–Douglas), and strictly decreasing in s when $\sigma > 1$ (substitutes).

(b) ***Unbounded upstream input.*** *When the upstream productivity gain grows without bound ($A \to \infty$), as under full automation of layer 1, the output gain converges to:*

$$\left.\frac{\tilde{y}}{y}\right|_{A\to\infty} = \begin{cases} (1-\theta)^{\frac{\sigma}{\sigma-1}} < \infty & \text{if } \sigma < 1 \text{ (complements)}, \\ \infty & \text{if } \sigma \geq 1 \text{ (Cobb–Douglas or substitutes)}. \end{cases} \tag{10}$$

Result (a) shows how the productivity gain is absorbed as it propagates through the hierarchy. The special case of Cobb–Douglas ($\sigma_s = 1$) provides a useful benchmark: each layer preserves a constant fraction θ of the previous layer's log gain ($R_s = \theta$ for all s), so $\ln G_s$ decays geometrically. With complements ($\sigma < 1$), R_s rises with s: the first layer above the shock absorbs a disproportionate share of the attenuation, and subsequent layers eat into the remaining gain much less. With substitutes ($\sigma > 1$), R_s falls with s: early layers preserve most of the gain, so a larger share reaches deeper layers, even though later layers attenuate more aggressively in relative terms.

Result (b) shows that with complements, even unbounded productivity gains in an upstream layer yield bounded gains. The reason is that with complements every input is essential, so even infinite AI input generates finite output if the human input remains finite. This result connects directly to the "weak links" framework in the macroeconomic growth literature (Kremer, 1993; Jones, 2011; Aghion et al., 2019; Jones and Tonetti, 2026). In that literature, when tasks in the economy are complements, automating some tasks has bounded effects on aggregate output—total production is constrained by the weakest link.[11] Our model applies this logic *vertically*: the "tasks" are the layers of the production hierarchy, and the weak link is the human-bottlenecked layer with the least AI capability.

The propagation results also relate to the literature on the pass-through of productivity shocks in production networks. In our setting, the hierarchical structure introduces nonlinearity: the impact of a shock at a given layer depends on input levels at that and higher layers, and on the elasticity of substitution between them. This connects to Baqaee and Farhi (2019), who extend Hulten's theorem (Hulten, 1978) to a nonlinear environment in which substitution elasticities shape how shocks propagate through the network.

[11] Jones (2026) shows that fully automating a subset of tasks raises output by at most a factor determined by the spending share of the tasks that are not automated; our bound (10) coincides with this result, with the share parameter θ in the role of Jones's spending share (see Appendix A.6).

3.5 Alternative Models and Extensions

So far, we have intentionally kept the model stylized to present the role of AI tools and to emphasize the most widely considered hypothesis linking task-level output to final output: complementarities and the weak-link hypothesis. However, there are alternative models of the software development production function that can generate similar attenuation patterns. We develop these alternatives in Appendix B, derive the corresponding pass-through expressions, and discuss how they differ from our baseline. In Section 6.3 we test the fit of these models to our empirical results.

One such model replaces complementarity with decreasing returns, as in a Cobb–Douglas technology $y_s = y_{s-1}^{\gamma_s} e_s^{\delta_s}$ with $\gamma_s + \delta_s < 1$. There, an overall productivity gain of A at a layer passes through as A^{γ_s}, so gains attenuate as they travel up the hierarchy even in the absence of complementarities, and decreasing returns deepen this attenuation. A second model combines complementarity with decreasing returns to scale, in which case attenuation is amplified because both channels operate at once.

A third model relaxes the assumption that developers hold their inputs fixed and instead allows them to reallocate inputs across layers. In this extension attenuation still arises, but is weaker than without input reallocation, since inputs are reallocated toward the layers with smaller AI productivity gains, which raises output relative to a fixed allocation. Reallocation can therefore be seen as a way of easing the bottlenecks, and it corresponds to the widely documented reorganization of production in response to new technologies (Brynjolfsson and Hitt, 2000; Bresnahan et al., 2002). While one can expect this to happen in the medium run, one could go further and consider changing the production process entirely to relieve the bottlenecks through complementary investments (Bresnahan and Trajtenberg, 1995; Brynjolfsson et al., 2021; McElheran et al., 2025). Since such changes have been shown to take a long time, we do not consider them as an extension.

4 Data and Summary Statistics

This section describes our data sources, how we identify adopters of each AI tool, the selection of the developer control pools, additional sample restrictions, and summary statistics.

4.1 Data Sources

Our analysis draws on three data sources: GitHub data on activity in public repositories, internal Microsoft data on the adoption and usage of AI coding tools available in GitHub Copilot, and data on four major software marketplaces. Detailed data collection procedures are provided in Appendix C.

GitHub Public Repository Data GitHub is the dominant platform for hosting and collaborating on software projects, with over 256 million developers and 517 million public repositories.[12] Repositories can be either public or private; our data include only activity on public repositories. While this restriction misses closed-source activity, open-source software is large and economically important in its own right (Wright et al., 2023; Hoffmann et al., 2024; Gortmaker, 2026). We later use our limited data on AI activity in closed-source repositories to assess the external validity of our public-activity results.

GitHub records detailed developer activity data—including commits, pull requests, and issues. We use these data first to identify the adoption of some of the AI tools we study (Claude Code and the async agents) from the traces these tools leave in public repositories. For these developers, we use the GitHub REST API to download complete commit and pull request histories from January 2020 through May 2026: 388 million commits and 46.9 million pull requests.[13] For each commit, we retrieve the author, timestamp, commit message, lines added and deleted, and the files changed. For each pull request, we retrieve the creator, creation date, requested reviewers, assignees, and branch name. These data allow us to construct a range of outcome measures, such as lines of code added, number of repositories touched, and files changed, which we aggregate into a weekly activity panel for all developers in our sample.[14]

Microsoft Data We use confidential data from Microsoft on GitHub Copilot usage. These data report, for all GitHub Copilot subscribers, the number of requests made to each tool in the Copilot subscription—autocomplete, the sync agent, and the async agent—between April and December 2025, together with subscription start dates from 2022. We use these data to construct adoption events for GitHub Autocomplete and GitHub Sync, and to validate our public adoption measures for the async agents.

Software Marketplace Data We obtain panel data from March 2020 through August 2026 on four software marketplaces: the Apple App Store, the Google Play Store, the Chrome Web Store, which we license through the APIs of SimilarWeb and chrome-stats.com, and SourceForge, which we scrape directly from the site. These data provide a time series of the apps available on each platform, together with app-level measures of usage such as

[12] As of August 2026, based on our own calculations described in Appendix C.

[13] An alternative data source is GHArchive, which is commonly used in the previous literature (Kreitmeir and Raschky, 2024; Gortmaker, 2026). Its advantage is that the data are available for bulk download and are not subject to GitHub API rate limits. However, our investigation of GHArchive revealed data quality issues after early 2025—including missing dates and incomplete commit activity coverage—making it unreliable for our paper. We therefore opt to download the data directly through the GitHub REST API.

[14] Because we obtain the data through API calls, our sample does not include coding activity that was later deleted by users. In addition, we collect activity information only from the main branches of the repositories to which developers contribute. However, in Section 6.2 we also analyze branch creation and deletion to capture activity on off-main branches.

Table 2: Construction of Adoption Events

Tool	Launch	Data source	Adoption event
Autocomplete			
GitHub Autocomplete	June 2022	Copilot subscriptions	Subscription start date
Sync agents			
GitHub Sync	Feb 2025	Microsoft internal usage	First use of Copilot agent mode
Claude Code	Feb 2025	Public GitHub	First commit to a `CLAUDE.md` file
Async agents			
GitHub Agent	May 2025	Public GitHub	First PR by `copilot-swe-agent[bot]`
OpenAI Codex	May 2025	Public GitHub	First PR on a `codex/` branch

Notes: The table summarizes how we identify adoption of each AI coding tool. "Launch" is the month the tool became publicly available; we include adoptions in the calendar year of launch. "Copilot subscriptions" and "Microsoft internal usage" are confidential data provided by Microsoft; "Public GitHub" data are collected through the GitHub REST API. For the async agents, adoption is measured only for developers who use the tool on a public repository.

installs, downloads, user counts, and ratings. Appendix C.7 provides the full details on data sources, collection method, and variables for each marketplace.

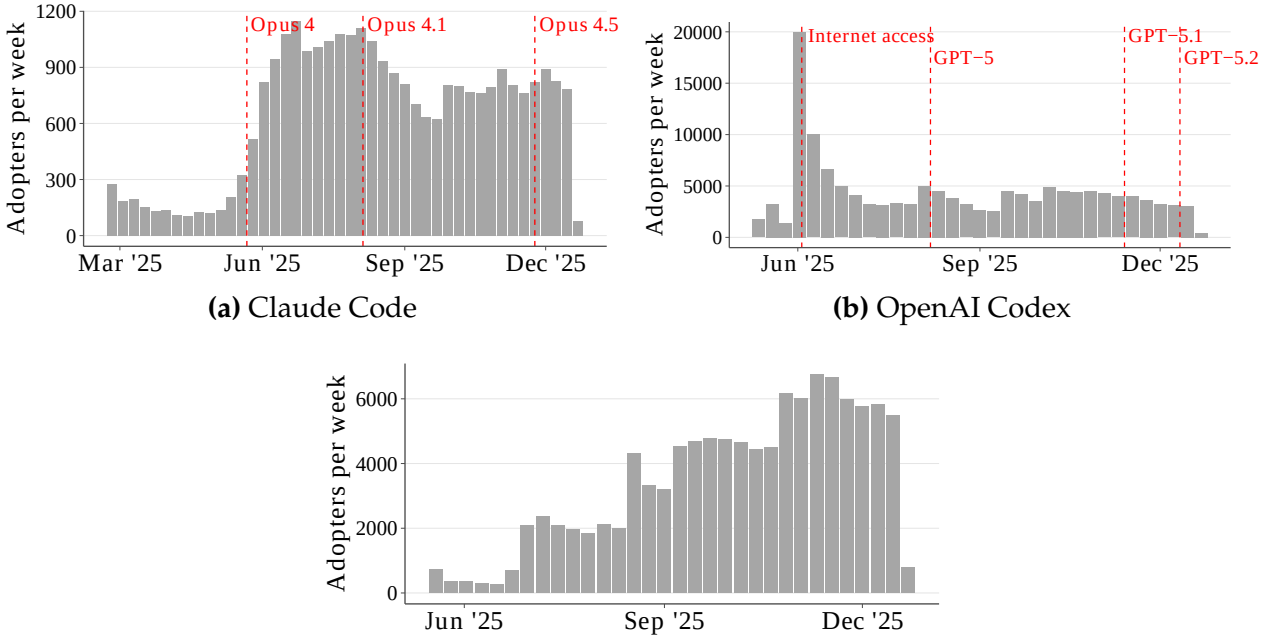
4.2 Construction of Adoption Events

Measuring AI adoption across different tools is challenging because usage data are typically proprietary. We observe internal telemetry for GitHub Copilot, but no equivalent exists for the other tools we study. We therefore combine these confidential records with publicly observable GitHub activity to identify adoption events across multiple tools and vendors. Table 2 summarizes the resulting adoption rules; we describe each in turn below.

GitHub Autocomplete We identify GitHub Autocomplete users using GitHub Copilot subscription data from 2022. Since autocomplete was the only AI tool available under the Copilot subscription at that time, we treat the subscription start date as the adoption date for GitHub Autocomplete.

Sync Agents For sync agents, we identify two tools: GitHub Sync and Claude Code. GitHub Sync adoption comes from internal Microsoft data: we use the first date each user invokes GitHub Copilot in agent mode as the adoption date. We identify Claude Code adopters by searching for `CLAUDE.md` files in public repositories. Claude automatically reads these configuration files when starting a conversation, making their presence a reliable indicator of Claude Code usage. For each user who has authored a commit to a `CLAUDE.md` file, we define the adoption date as the date of their earliest such commit. Figure 3(a) plots the resulting Claude Code adoption events, which ramp up sharply around the release of Opus 4 in late May 2025 and continue throughout our sample.

Figure 3: Adoption of AI Coding Tools over Time



(a) Claude Code

(b) OpenAI Codex

(c) GitHub Agent

Notes: Each panel plots the weekly count of new adopters for the corresponding tool over our sample window. Vertical dashed red lines mark the release dates of frontier models behind each tool: Anthropic Opus 4 (May 22, 2025), Opus 4.1 (Aug 5, 2025), and Opus 4.5 (Nov 24, 2025) in Panel 3(a); OpenAI GPT-5 (Aug 7, 2025), GPT-5.1 (Nov 13, 2025), and GPT-5.2 (Dec 11, 2025), together with the date Codex agents were given internet access (Jun 3, 2025), in Panel 3(b). Panel 3(c) omits such markers as the underlying model behind GitHub Agent has not been publicly versioned during our window. We omit the corresponding distributions for GitHub Autocomplete and GitHub Sync as those data are considered sensitive.

Async Agents Unlike sync agents, async agents interact directly with the version-control system and hence leave unambiguous traces. Using data from GitHub repositories, we retrieve all pull requests created by the Copilot coding agent bot for the GitHub Agent, and all pull requests opened on Codex-created branches for Codex.[15] From these pull requests, we identify the associated human users through co-author information in commit messages, requested reviewers, creators, or assignees, and use the earliest such pull request as their adoption date. Figures 3(b) and 3(c) plot the resulting adoption events for the two async tools. While Codex adoption jumps sharply with the introduction of internet access in early June 2025 and remains roughly stable thereafter, GitHub Agent adoption is concentrated noticeably later in the sample, accelerating through the late summer and fall. This timing gap suggests that the two adopter pools may differ, so cross-tool differences in treatment effects could partly reflect who adopts rather than the tools themselves.

Async agents come bundled with sync capabilities: a developer with access to the

[15]The GitHub Agent's pull requests are authored by the bot account `copilot-swe-agent[bot]`; Codex pull requests are opened on branches with the prefix `codex/`.

GitHub Agent or Codex can also use the same product interactively, and Microsoft internal data show that async adoption is associated with roughly a fivefold increase in sync usage, with only a small share of developers using the async agent exclusively. We therefore treat async adoption as the joint adoption of sync and async capabilities, and we separate the two effects in our empirical analysis: agent-authored commits can come only from the async agent, while changes in the developer's own commits reflect sync use. Accordingly, in our per-tool estimates, the GitHub Agent adopters' human-authored activity enters the GitHub Sync sample, since both reflect use of the same product's sync mode.

4.3 Selecting Adopters and Summary Statistics

We impose three restrictions on the adopters we study. First, we include only adoptions in the calendar year of the tool's release: for autocomplete, this rules out contamination from tools that arrived later, such as chatbots, and for the tools launched in 2025, it ensures that controls are drawn from a year before agentic tools were widely adopted.[16] Second, we require that we can retrieve an adopter's commit history: this drops accounts that have been deleted or suspended, accounts at bot scale (more than 100,000 commits in the window or 1,000 in a day), and adopters with no public commits, and reduces the sample from 752,917 to 669,107 adopters. Third, we require at least one week with positive commits at or before week −11 relative to adoption, when our event study begins, so that we study developers who were already coding on GitHub rather than accounts created to try out a tool; this leaves 521,329 adopters. Appendix Table OA-4 reports how many adopters remain after each of these restrictions, tool by tool.

Table 3 reports pre-adoption summary statistics for adopters after the sample restrictions: weekly averages over the eleven weeks before adoption for the six outcomes spanning the production hierarchy. We highlight three features. First, activity is highly dispersed within each sample: for commits, the standard deviation is about 4 times the mean, and the 5th percentile is zero, reflecting a mix of full-time developers and sporadic contributors. This dispersion is why we normalize outcomes by each developer's own pre-period activity in our event study, as described in Section 5.2. Second, autocomplete adopters are less active on GitHub than agent adopters, averaging 1.9 commits per week against 3.5 for the sync sample. This may partly reflect lower overall activity on GitHub in the earlier period, and partly that some GitHub Copilot subscribers are not active on GitHub, coding mostly elsewhere. Finally, releases are rare in any given week; we return to this when interpreting attenuation across the production hierarchy.

[16] The exact dates of these windows are given in Appendix Table OA-3.

Table 3: Pre-Adoption Summary Statistics for Adopters

Sample	Lines Changed	Distinct Files	Commits	PRs Created	Distinct Repos	Releases
GitHub Autocomplete	9,773.58	9.04	1.85	0.17	0.32	0.02
(sd)	(160,842.64)	(31.82)	(7.24)	(0.82)	(0.71)	(0.18)
[p5, p95]	[0.00, 18,452.67]	[0.00, 47.22]	[0.00, 9.00]	[0.00, 0.82]	[0.00, 1.45]	[0.00, 0.09]
Pooled Sync	15,898.31	15.68	3.46	0.38	0.45	0.06
(sd)	(270,369.20)	(113.07)	(14.13)	(1.99)	(1.02)	(0.34)
[p5, p95]	[0.00, 36,699.14]	[0.00, 72.82]	[0.00, 15.09]	[0.00, 1.91]	[0.00, 1.82]	[0.00, 0.27]
Pooled Async	18,033.49	16.74	3.66	0.38	0.47	0.06
(sd)	(315,960.51)	(133.78)	(13.34)	(1.94)	(1.00)	(0.36)
[p5, p95]	[0.00, 45,049.90]	[0.00, 76.82]	[0.00, 15.91]	[0.00, 2.00]	[0.00, 1.82]	[0.00, 0.27]

Notes: This table reports pre-adoption summary statistics across all outcomes we consider and the three main samples of adopters. Statistics are computed over all adopters satisfying the requirements discussed in Section 4.3 (and correspond to the "History floor" column of Table OA-4), before matching. "Pooled Sync" pools adopters of GitHub Sync, GitHub Agent, Claude Code, and OpenAI Codex, as in the sync event studies. "Pooled Async" pools adopters of GitHub Agent and OpenAI Codex, as in the async event studies. Each column reports a weekly outcome averaged over the pre-adoption period (weeks −11 to −1 relative to adoption). For each sample, the first line is the mean across adopters, followed by the standard deviation in parentheses and the 5th and 95th percentiles in brackets.

4.4 Construction of the Control Pool

We also collect data on a broad set of GitHub developers from which we later draw matched controls. Downloading commit histories for every GitHub account is infeasible under API rate limits, so we use GHArchive, a public log of all events on public repositories, to screen for active accounts. We include every account with at least twenty push events over 2021–2025 (13.4 million accounts) together with a stratified random sample of 10 million less active accounts. Of these, 20.8 million accounts still exist and have at least one public commit; these developers form the control pool, and we download their complete commit and pull request lists—2.7 billion commits and 201 million pull requests.

We collect commit details, which requires one API call per commit, only for the developers matched to an adopter in Section 5.1 (344 million commit records), from which we construct the same outcome variables as for the treated group. Appendix C.3 describes the construction in detail and Appendix Table OA-5 reports the counts at each stage.

5 Productivity Effects Across Generations of Tools

Our empirical analysis examines the productivity effects of AI coding tools along two dimensions: across successive generations of tools, and across stages of the software production hierarchy. We organize the analysis as follows. We first describe how we match adopters to controls and set out our event-study specification. We then use weekly

commits, our primary task-level productivity measure, to estimate the productivity effects across the three generations of tools. Then, in Section 6, we move beyond commits and examine outcomes at each layer of the production hierarchy. We interpret these results through the lens of our production model to understand how large task-level productivity effects translate into gains in final output, and to explore the underlying mechanisms.

5.1 Matching Strategy

For each coding tool, we perform a matched event study, pairing each adopter with a developer from the control pool who never adopts the tool. We match on pre-adoption activity because adoption is not random in time. Developers adopt a new tool when they are actively coding, so adopters are, on average, in the middle of an activity spell at adoption, and a developer who is unusually active this week tends to remain unusually active for some weeks afterward. Comparing adopters to controls drawn at an arbitrary time would therefore attribute the tail of the spell to the tool. Matching on activity at and before adoption is how we match on the trajectory that follows.

Concretely, each adopter is matched to a control who (i) made at least one commit and pull request in the pseudo-adoption week—the strongest activity signal available to us—and (ii) is closest in total commits over the eleven preceding weeks[17], one-to-one and without replacement.[18] Because there is judgment in choosing the activity signal, we also re-estimate the event study under weaker activity requirements—only a commit, or only a pull request, in the (pseudo-)adoption week—and obtain similar estimates (Appendix Table OA-11). We also conduct placebo tests using non-AI tools, discussed in Section 5.5.

Table 4 compares the resulting treated and matched control units for each event study. Because our donor pool covers developers at any given activity level, matching on pre-period commits ensures that median commit activity is identical across the two groups. Matching on commits also brings the other pre-period outcomes close, though not perfectly: treated developers change more lines per week, while control developers open more pull requests and touch more repositories. Where the two groups differ, controls are if anything the more active—the intended consequence of requiring controls to have both a pull request and a commit in the pseudo-adoption week. Because our specification includes a matched-pair fixed effect, these level differences do not enter the estimates; what matters is that pre-trends are flat, which we verify for every outcome below.

[17] As we match on *total* commits and not the week-by-week pattern, pre-trends remain informative.

[18] For tools whose adoption we observe on GitHub (Claude Code, the async agents, and Docker), the adoption event is itself public activity. For tools observed through Microsoft subscription or usage records (autocomplete, GitHub Sync, and GitHub Pro), which need not coincide with public activity, we impose the same commit-plus-pull-request requirement on adopters, so that adopters and controls are selected on the same signal; estimates are similar when this requirement is imposed on all adopters (Appendix Table OA-11).

Table 4: Comparison of Treated and Matched Control Units

	GitHub Autocomplete	GitHub Sync	Claude Code	GitHub Agent	OpenAI Codex
Median Weekly Lines Changed					
# Treated Units	14,191	5,898	19,902	61,774	50,726
Treated Group	1,917	884	1,299	1,226	743
Matched Control Group	1,504	716	865	776	442
Median Weekly Files Touched					
# Treated Units	14,221	5,927	19,980	62,134	51,159
Treated Group	16.18	8.82	9.36	9.09	5.09
Matched Control Group	15.18	9.45	10.64	9.45	6.00
Median # Weekly Commits					
# Treated Units	14,283	5,968	20,161	62,801	51,919
Treated Group	4.09	2.18	2.36	2.09	1.27
Matched Control Group	4.09	2.18	2.36	2.09	1.27
Donor Pool	0.75	0.77	0.76	0.76	0.76
Median Weekly PRs Created					
# Treated Units	10,629	4,008	9,084	27,947	14,294
Treated Group	1.00	0.73	0.45	0.55	0.36
Matched Control Group	1.09	0.82	1.09	1.00	0.91
Median Weekly Repos Touched					
# Treated Units	14,225	5,930	19,986	62,169	51,195
Treated Group	0.91	0.55	0.45	0.45	0.27
Matched Control Group	0.91	0.64	0.64	0.64	0.45
Median Weekly Releases					
# Treated Units	1,609	572	1,775	4,740	1,840
Treated Group	0.36	0.27	0.27	0.27	0.18
Matched Control Group	0.36	0.27	0.36	0.36	0.27

Notes: This table presents median features of treated units and the control units matched to them, separately for each productivity event study we conduct. For each outcome variable, "# Treated Units" reports the number of treated users entering that regression, after dropping developers with no pre-period activity in that outcome and the top 1% of developers by pre-period volume. "Treated Group" and "Matched Control Group" report the median of user-level average pre-period outcome values (weeks −11 to −1 relative to the event). For the "Donor Pool" row under Commits, we report the median of user-level average weekly commits for the full calendar window of each event study. "GitHub Sync" covers the adopters identified from Microsoft usage data; the GitHub Sync event studies of Tables 5 and 6 additionally pool the human-authored activity of the GitHub Agent adopters shown in their own column.

When matching control users to treated users, we take the control group's outcomes from exactly one year earlier, because contemporaneous non-adopters are not a clean comparison group, for two reasons. First, and most importantly, many developers use AI tools without leaving a trace in public repositories, so a contemporaneous control group is contaminated by treatment. Second, in a period of widespread adoption, non-adoption is

itself informative about a developer's future: those who have adopted no AI tool by 2025 are disproportionately likely to be reducing their activity. Contemporaneous non-adopters may therefore be on a declining trajectory relative to adopters, a violation of parallel trends that matching on pre-adoption activity does not address. Drawing controls from the year before the tool was widely available avoids both issues: as no developer in the control pool could have adopted, the pool is neither contaminated nor selected on the decision not to adopt. Nevertheless, we re-estimate the event study using a contemporaneously matched control group as a robustness check in Appendix E.

Appendix F formally develops the identification argument, showing that the primary new identifying assumption required for this strategy (relative to classic difference-in-differences) is the absence of calendar-time (aggregate) shocks after the last adoption event. Appendix D provides the full details of the matching procedure.

5.2 Empirical Specification

Using this sample, we estimate a generalized difference-in-differences specification:

$$Y_{it}^{T} - Y_{it}^{C} = \sum_{k=-11;k\neq-1}^{30} \beta_k \cdot \mathbf{1}\{t - t_i^* = k\} + \alpha_i + \varepsilon_{it}, \tag{11}$$

where Y_{it}^{T} is the weekly outcome of treated developer i in week t, normalized by her pre-period mean[19]; Y_{it}^{C} is the analogous outcome for the matched control; t_i^* is the (pseudo-)adoption week; and α_i is a matched-pair fixed effect. After estimation, we re-normalize the pre-period coefficients to have zero mean, so each β_k is interpreted as the deviation from the average pre-period level. The coefficients β_k thus trace the dynamic treatment effect at event time k, expressed as proportional changes relative to the pre-period baseline. Standard errors are clustered at the developer level (developer-by-tool when pooling tools). We estimate this specification for several outcome measures, including lines of code, files, commits, pull requests, repositories, and releases; in all specifications, we winsorize the dependent variable (the treated–control difference of normalized outcomes) at the 1st and 99th percentiles. Appendix D gives the full specification, including pre- and post-window indicators.

Because outcomes are normalized by each developer's pre-period mean, β_k is the

[19]Dividing by the pre-period mean requires the developer to be active in the estimation outcome between weeks −11 and −1. For most outcomes this imposes little additional restriction, since we already condition on developers who are active on GitHub (Section 4.3). However, some outcomes, such as releases, are quite sparse; for these outcomes we drop developers with no pre-period activity of that type. While this creates some differences in the estimation sample across outcomes, we conduct a robustness check restricting the sample to developers with pre-period activity in every outcome (Appendix E.2).

average of individual percentage changes, as with a log outcome. This differs from a level effect divided by mean productivity when effects vary with baseline activity, as they do here: if a developer with 1 commit per week rises to 2 and one with 100 rises to 110, the average percentage change is 55%, while the average level change (5.5 commits) is about 11% of the overall mean.[20] We prefer this to logs, which would require dropping the many developers with zero commits in some week, and present it as our main specification, leaving level estimation as a robustness check.

Our main estimates pool AI tools within each generation, following the decomposition of Section 4.2: the async-agent effect is estimated from async adopters' agent-authored activity, and the sync-agent effect from their human-authored activity pooled with that of the sync-agent adopters. For agent-authored commits, the control's outcome and the adopter's pre-period outcome are zero by definition, so the same specification reduces to an interrupted time series; because the outcome's own pre-period mean is zero, we normalize by the developer's pre-period mean of total commits. We report product-specific effects in a subsequent heterogeneity analysis.

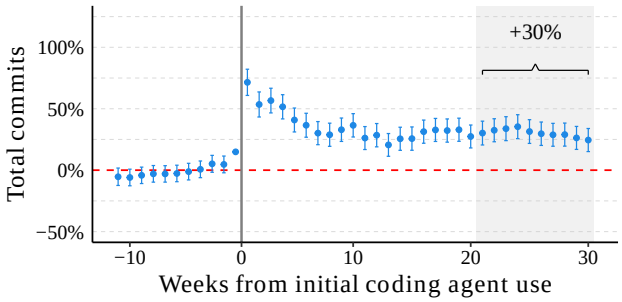
5.3 Results: Task-Level Productivity

We now present results on how task-level productivity differs across generations of AI tools. Unlike settings such as essay writing or customer service, where output can be scored directly, there is no consensus measure of software-development productivity (Petersen, 2011; Meyer et al., 2014; Ko, 2019; Forsgren et al., 2021). Given this, we follow the AI productivity literature in settings such as coding (Cui et al., 2026) and customer service (Brynjolfsson et al., 2025) and estimate task-level productivity: holding the unit of work roughly fixed, we ask how much more code a developer produces. We use weekly commits as our outcome. Commits are a natural discrete unit of coding work, are widely tracked as a developer-activity measure in the software-engineering literature (Forsgren et al., 2021; DORA, 2024), and are observed at high frequency in our data. In Section 6, we examine other outcome variables to understand how AI affects different stages of the production process and final output.

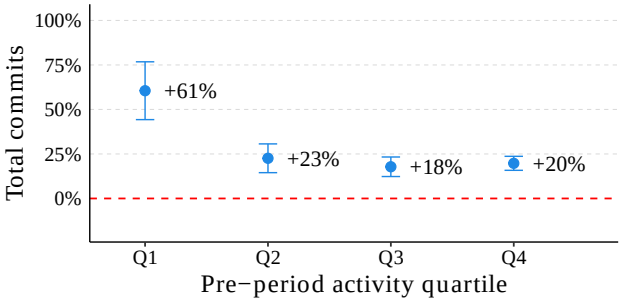
Figure 4 presents the event studies; Table 5 reports the corresponding estimates pooled into weeks 1–10, 11–20, and 21–30 after adoption. The effect is shown separately for each generation: autocomplete, sync agents, and async agents. The top row shows users who adopted GitHub Copilot in 2022, when only AI-based code completion was available. Our

[20]The pre-period mean is estimated from 11 weeks and is therefore noisy; by Jensen's inequality, dividing by a noisy mean may bias the normalized outcome upward. In practice this bias is small: we estimate a level specification, which does not divide by the pre-period mean (Table OA-9), and another specification where we calculate the pre-period mean from 52 weeks, and obtain similar results (Appendix Table OA-12).

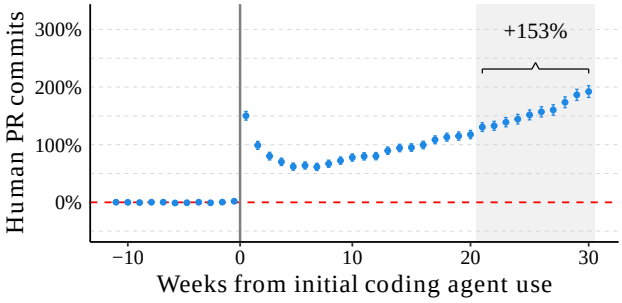
Figure 4: Task-Level Productivity Effects of AI Coding Tools



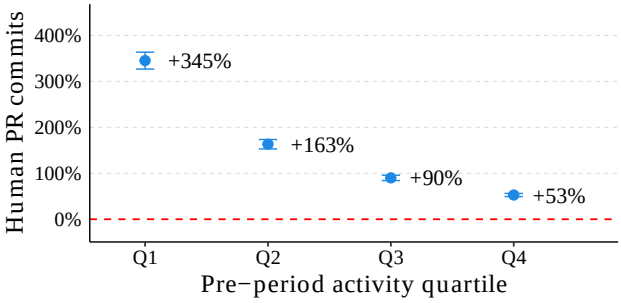
(a) Autocomplete: Event Study



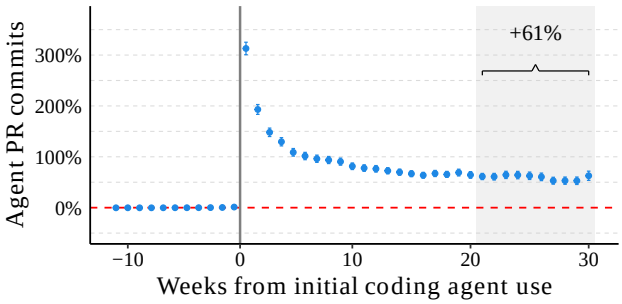
(b) Autocomplete: Heterogeneity



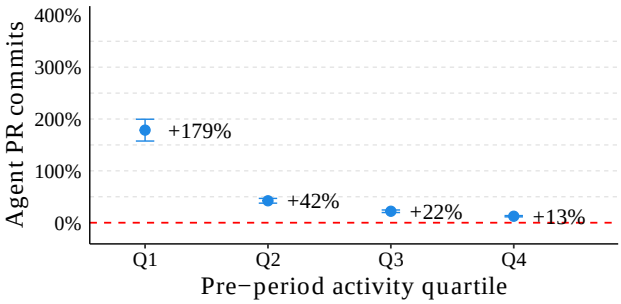
(c) Sync Effect: Event Study



(d) Sync Effect: Heterogeneity



(e) Async Effect: Event Study



(f) Async Effect: Heterogeneity

Notes: The left column presents matched event-study (Panels 4(a) and 4(c)) or interrupted time series (Panel 4(e)) estimates of the effects of generative AI coding tools on commits; the right column shows treatment effects by pre-period activity quartile (weeks 21–30). Panel 4(a) compares users who subscribed to GitHub Copilot in 2022 to a matched control group active a year prior. Panel 4(c) pools adopters of sync agents (Claude Code, GitHub Sync, GitHub Agent, and OpenAI Codex). Panel 4(e) shows the effect on agent-authored commits for async agent adopters (GitHub Agent and Codex). Outcomes are measured relative to each developer's pre-period mean. The first post-adoption week is omitted as large transitory effects distort the scale; the omitted coefficients are 102%, 236%, and 822% for Panels 4(a), 4(c), and 4(e), respectively. In the right column, quartiles are assigned based on the average of treated and control developers' mean pre-period commits, to avoid regression to the mean.

estimate in Figure 4(a) shows a 30% increase in the number of commits. The pre-trend is almost perfectly flat, supporting the parallel trends assumption. As shown in Figure 4(b), the effect is larger for less active developers, though it remains substantial throughout the activity distribution (from 20% for the most active to 61% for the least active). Our estimates are consistent with prior field-experimental estimates of a 26% (SE: 10%) task-level productivity gain from GitHub Copilot autocomplete (Cui et al., 2026), validating our design against experimental evidence.

Moving on to sync agents in the middle row, we again find flat pre-trends and a sizable effect immediately upon adoption (Figure 4(c)). While the number of commits temporarily increases sharply—likely due to an experimentation phase—the effect rapidly decays before rising again to an eventual 150% productivity gain—about 5 times the effect of autocomplete. The heterogeneity pattern (Figure 4(d)) mirrors that of autocomplete: effects are largest for less active developers and decrease monotonically as developers become more active, though even for the most active developers the effect remains around 53%. We caution that, while these heterogeneity results are consistent with the prior experimental literature (Peng et al., 2023; Brynjolfsson et al., 2025; Cui et al., 2026), there are theoretical reasons to expect the direction of the heterogeneity to flip as tools improve (Benzell and Myers, 2026).

Finally, the bottom row shows the effect of adopting async agents on agent-written commits (Figure 4(e)). Agent-authored commits are zero by definition before adoption, and they rise to about 60% of a developer's baseline commits after adoption. As with the other tools, effects spike initially before settling at moderate but significant levels, again larger for less active developers (Figure 4(f)).

In Table 5, we break out the sync and async effects by product; the corresponding per-tool event studies are in Appendix Figure OA-4. We see large and statistically significant effects across all coding tools, suggesting that our results are not driven by the selection of a particular tool. However, the magnitudes vary by tool: the long-run sync effect is 296% for Claude Code, 126% for GitHub Sync, and 119% for OpenAI Codex. These differences can reflect either differential productivity effects of the tools themselves or differences in the population of developers who adopt each tool. For example, Claude Code's notably larger sync effect may reflect its adopters: the tool runs from the command line and was adopted early in our sample (Figure 3), both markers of sophisticated developers who might benefit more from any tool. Our empirical strategy cannot distinguish between these mechanisms.

Taken together, our results establish that generative AI coding tools have substantial effects on developer task-level productivity, with effects that grow across successive gener-

Table 5: Task-Level Productivity Effects of AI Coding Tools

	% Change in Number of Commits		
	Weeks 1–10	Weeks 11–20	Weeks 21–30
Autocomplete	44.1	28.3	30.2
	(2.6)	(2.5)	(2.7)
Sync Effect	81.3	98.5	153.2
	(1.9)	(2.0)	(2.7)
Claude Code	206.5	211.1	295.5
	(5.7)	(6.2)	(7.7)
GitHub Sync	58.5	86.7	126.3
	(2.2)	(2.4)	(3.2)
OpenAI Codex	61.3	66.9	118.8
	(3.7)	(3.8)	(4.6)
Async Effect	138.5	70.2	60.7
	(2.8)	(2.1)	(2.4)
GitHub Agent	92.8	59.0	62.4
	(2.3)	(2.0)	(2.5)
OpenAI Codex	196.0	83.9	63.3
	(5.5)	(3.9)	(4.0)

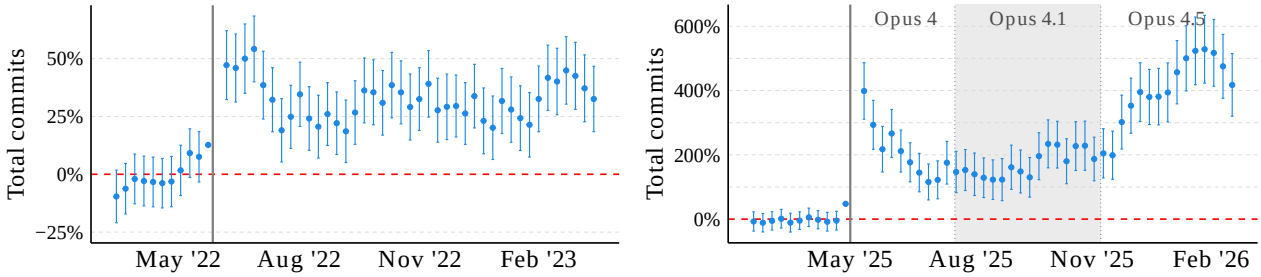
Notes: This table reports pooled event-study coefficients for the effect of adopting each generation of AI coding tools at three horizons. Estimates for weeks 1–10 may be inflated by transitory experimentation or activity bias; our preferred estimates are those for weeks 21–30. Standard errors, clustered at the developer level (developer-by-tool for the pooled rows), in parentheses. The "GitHub Sync" row pools adopters identified from Microsoft usage data with GitHub Agent adopters, whose human-authored commits also reflect sync use (Section 4.2).

ations. Autocomplete increases commits by roughly 30%; the cumulative effect including sync agents rises to approximately 180%; and additional use of async agents raises it to 240% through agent-authored commits. These effects are positive and statistically significant for every tool, and persist over at least 30 weeks. The significantly smaller effects of autocomplete relative to agents are consistent with the nature of these tools as discussed in Section 3: autocomplete simply augments the developer's own input when writing code, whereas agents can lead to partial or full automation.

5.4 Results: Dynamic Treatment Effects

The treatment effects in Figure 4(c) showed a clear increase over event time, with effects in weeks 21–30 noticeably larger than in earlier weeks. This pattern admits two interpretations: a genuinely dynamic treatment effect that accumulates with time since adoption (e.g., as users invest in complementary capital or learn to use the tool more effectively), or a calendar-time effect that operates on all users at each point in time (e.g., as the frontier models powering these tools improve). The two are difficult to disentangle in the pooled sample, where adoption dates are spread across calendar time. To make progress, we restrict attention to the subset of developers who adopted within the first three weeks of

Figure 5: Task-Level Productivity Effects over Calendar Time



(a) GitHub Autocomplete **(b)** Claude Code

Notes: Each panel presents the matched event-study estimate of the effect of adopting the corresponding tool on weekly commits, restricted to the subset of developers who adopted in the first three weeks after the tool's release. The empirical specification is otherwise identical to the matched event study in Figure 4(c). Panel 5(a) shows GitHub Autocomplete adopters in mid-2022; to our knowledge, the underlying model was not updated until February 2023 and was not significantly updated until November 2023 (when it was upgraded to GPT-4), so the entire window in this panel reflects a single model generation. Panel 5(b) shows Claude Code adopters in mid-2025; the cohort is anchored on the date Claude Code became generally available (May 22, 2025, coinciding with the Opus 4 launch) rather than the February 2025 research preview. The shaded regions mark the eras during which successive Anthropic Opus models (Opus 4, Opus 4.1, Opus 4.5) were the frontier release, with era boundaries corresponding to each model's public release date.

each tool's release. Within this fixed cohort, event time maps directly onto calendar time, so we can read calendar-time effects off the horizontal axis. Figure 5 reports the results.

Panel 5(b) shows that the increasing event-time pattern for Claude Code is preserved within the early-adopter cohort, and aligns with the release of successive Opus models: the effect declines and stabilizes through the Opus 4 and Opus 4.1 eras, then rises sharply once Opus 4.5 becomes the frontier model. Panel 5(a) reinforces this reading by contrast: GitHub Autocomplete showed no increasing pattern in event time (Figure 4(a)), and the early adopters likewise show no increase in calendar time after the initial spike. This is consistent with the fact that the cohort adopted in mid-2022, before any non-trivial update to the underlying model—the first came in February 2023, with a more substantial upgrade to GPT-4 in November 2023, well outside our window. Together, these patterns suggest that the increasing long-run effects we estimate for sync agents are driven primarily by improvements in the frontier models that power them, rather than by within-developer learning over time.[21]

5.5 Robustness Checks

Our analysis relies on observational data, which raises concerns about identifying a causal effect. We summarize several robustness checks here that support a causal interpretation of our estimates; full details are in Appendix E.

[21] Consistent with this interpretation, Sarkar and Melas-Kyriazi (2026) document a 44% increase in weekly agent messages on Cursor following the late-2025 releases of Claude Opus 4.5 and GPT-5.2, with the response concentrated in more complex tasks.

Commit quality and size. One concern is that the increase in commits may not reflect an increase in task-level output. This could happen for two main reasons. First, AI may reduce the quality of commits: the additional commits may consist of buggy code and subsequent fixes, reflecting rework rather than new output. Second, the size of commits may shrink: if AI leads developers to split a given amount of work into more, smaller commits, commit counts would rise without an increase in underlying output. We test the first hypothesis by classifying commits into bug fixes and feature work based on commit messages, and the second by re-estimating our event study with lines per commit and files per commit as outcomes. The share of bug-fix commits relative to feature commits remains stable after adoption (Appendix Figure OA-16), and we find no evidence of a decline in commit size (Table 7). Together, these results indicate that the increase in commits reflects a real increase in task-level output.

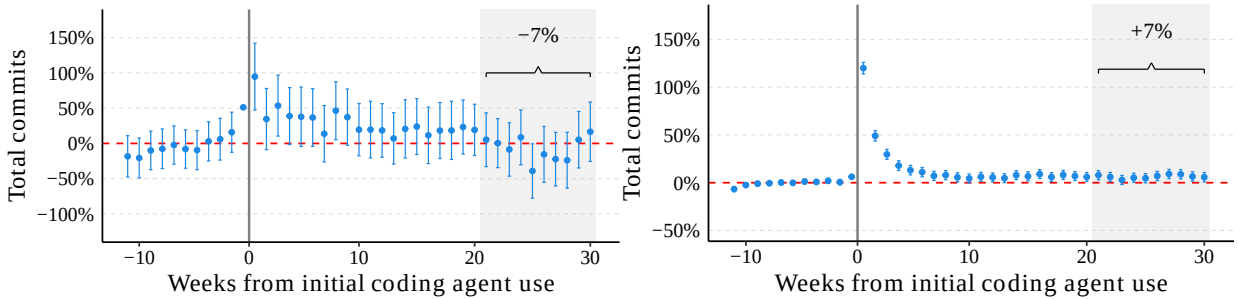
Activity bias. A second concern is that our estimates capture *activity bias*—periods of heightened engagement that coincide with adoption—rather than a true productivity effect. Two pieces of evidence argue against this. First, we examine two non-AI coding tools: GitHub Pro, a paid subscription without AI features, and Docker, a containerization tool that may itself raise productivity and so serves as an upper bound on activity bias.[22] Figure 6 shows that both generate large short-run spikes that decay rapidly, with GitHub Pro converging to no long-run effect and Docker stabilizing around 7%, against 153% for AI coding tools. Second, if our estimates were driven by activity bias, they would dissipate over time, as they do in the placebos within roughly 10 weeks. Instead, Table 5 shows that the AI treatment effects remain stable over the 30-week post-adoption window.

Robustness to empirical specification. To assess the robustness of our specification that scales outcomes by the pre-period mean, we also estimate the event study in levels. Specifically, we estimate a separate level regression for each pre-period-mean decile—allowing the effect to differ across baseline-activity groups—and then average the decile-specific estimates. The results in Table OA-9 show that this alternative approach yields effects closely matching our pooled normalized estimates. We further examine robustness to other specification choices, such as the winsorization threshold and the sample restrictions, and find similar results in Appendix Table OA-7 and Appendix Figures OA-12 and OA-13.

Robustness to matching. To assess how our results depend on the matching procedure, we re-estimate the event study under alternative activity requirements. We require only a

[22] GitHub Pro adoption is identified from Microsoft subscription data in 2022, in the same way as autocomplete. Docker adoption is identified from a developer's first commit to a `Dockerfile`, in the same way as Claude Code adoption is identified from `CLAUDE.md` files; we include Docker adoptions from 2022 through 2025 to have sufficient statistical power.

Figure 6: Placebos: Productivity Effects of Other Coding Tools



(a) GitHub Pro **(b)** Docker

Notes: This figure presents estimates from two matched event studies assessing the productivity effects of non-AI coding tools. Panel 6(a) compares users who subscribed to GitHub Pro in 2022 to a matched control group of users who authored a pull request and a commit in the same week one year earlier; Panel 6(b) does the same for users who adopted Docker between 2022 and 2025. Outcomes are measured relative to a developer's average number of commits in the pre-period (i.e., the average pre-period outcome in both groups is one by construction). While adopters of both tools show a large temporary increase in activity—consistent with either a true short-run productivity effect or activity bias—these effects quickly diminish. In the long run, we find no effect from GitHub Pro and only a small effect from Docker. The first post-adoption week is omitted as large transitory effects distort the scale; the omitted coefficients are 244% and 340% for Panels 6(a) and 6(b), respectively.

commit, or only a pull request, in the (pseudo-)adoption week of every adopter and every control; we require both of every adopter, including those whose adoption we observe on GitHub; and we additionally require matched pairs to have created their accounts in the same year. Appendix Table OA-11 and Appendix Figure OA-11 report the results, which are similar to our baseline estimates.[23] We also re-match on longer pre-adoption windows, using total commits over the 22, 32, 42, or 52 weeks preceding adoption instead of 11, and again find similar estimates (Appendix Table OA-12).[24]

Using the previous year as controls. As explained in Section 5.1, we match control developers to their activity one year before adoption, since contemporaneous controls are likely to be using AI tools in ways we do not observe. We nevertheless re-estimate the event study with contemporaneous matching (Appendix Table OA-11 and Appendix Figure OA-11, Panels OA-11(a) and OA-11(b)). As discussed earlier, this specification yields smaller estimates, but only by 21% on average, so even the contemporaneous specification implies statistically and economically significant productivity effects of AI. We also provide a formal discussion in Appendix F of the conditions under which this design identifies the treatment effect, and argue that these conditions are likely to hold in our setting.

[23] We run these variants for commits and pull requests only, since downloading the remaining outcomes for the full control pool is prohibitively time- and compute-intensive. We do not condition on account creation year in the main specification because doing so lowers the matching rate.

[24] We nevertheless prefer the eleven-week window: activity comes in spells, and commits over the past year are a weaker signal of whether a developer is currently in an active spell than commits over the past quarter.

External validity: public vs. private repositories. Our adoption measure relies on publicly observable activity, raising the concern that we miss developers who use AI tools exclusively on private repositories. We provide two pieces of evidence that our sample nevertheless captures a large share of total activity on GitHub. First, using internal GitHub data on async agent usage in November 2025, we find that our public-repository measure identifies 24.2% of all async agent users and captures 72.9% of these adopters' usage activity. Second, using GitHub repository identifiers, we estimate that 38% of all repositories ever created are public. Evidence from proprietary enterprise data also points in the same direction as our public-repository results (Chen and Stratton, 2026).

Robustness to activity deletion. Because we collect our data through the GitHub API, activity that developers have since deleted does not appear in our sample. To gauge how much this matters, we measure deletion rates directly at the user and repository level, and find annualized deletion rates of 0.4% for user accounts and 5.8% for repositories. Deletion therefore affects only a small fraction of the activity we observe.

6 Productivity Effects Across the Production Hierarchy

Having documented large task-level productivity effects that grow across successive generations of AI tools, we next ask how these task-level gains translate into final output. We do so by examining final-output measures directly—the number of repositories and releases—and we also use our model as a guide to understand the underlying mechanics: if task-level gains fail to translate into final output as the weak-link hypothesis predicts, we should see the effect decrease as we move to higher levels of task aggregation. To test this, we look at the effect of AI tool adoption on outcomes at each layer of the production hierarchy: lines of code, files, commits, pull requests, repositories, and releases.

6.1 How Does Task-Level Productivity Affect Final Output?

In Table 6, we find a pattern consistent with this prediction. The columns correspond to progressively higher layers of the production hierarchy, from lines of code through releases. Across all three generations of tools, the effects are largest for lines of code, smaller for files, commits, and pull requests, and smaller still for repositories and releases.

Starting with autocomplete, the effects decline monotonically across layers: from 234.3% for lines of code, to 30.2% for commits, and to only 9.0% for releases. This is precisely the pattern predicted by Corollary 1: because autocomplete operates only at the code-writing layer (Section 3), its gains must pass through every higher layer, and each layer, bottlenecked by human input, absorbs part of the gain. Moving to sync agents, the same monotonic pattern holds at larger magnitudes: the effect on lines of code is 957.5%,

Table 6: Attenuation of Productivity Effects Across the Production Hierarchy

	Outcomes (Weeks 21–30, %)					
	Lines Chg	Files Touched	Commits	PRs	Repos Touched	Releases
Autocomplete	234.3	50.0	30.2	18.3	11.9	9.0
	(17.9)	(4.2)	(2.7)	(2.2)	(1.5)	(5.0)
Sync Effect	957.5	265.0	153.2	86.0	47.3	19.8
	(23.2)	(4.7)	(2.7)	(2.1)	(0.9)	(2.9)
Claude Code	1343.4	408.7	295.5	178.9	87.5	39.4
	(52.6)	(11.8)	(7.7)	(6.6)	(2.3)	(6.8)
GitHub Sync	935.8	240.5	126.3	65.5	46.9	17.3
	(27.5)	(5.8)	(3.2)	(2.3)	(1.2)	(3.8)
OpenAI Codex	777.8	224.1	118.8	61.2	28.5	5.2
	(44.9)	(8.4)	(4.6)	(3.8)	(1.4)	(5.5)
Async Effect	1254.9	83.3	60.7	68.5	19.2	—
	(75.3)	(3.5)	(2.4)	(3.4)	(0.3)	
GitHub Agent	1292.2	92.8	62.4	73.2	22.6	—
	(104.4)	(5.0)	(2.5)	(3.2)	(0.5)	
OpenAI Codex	1243.4	77.4	63.3	72.4	17.1	—
	(109.8)	(4.9)	(4.0)	(7.0)	(0.5)	

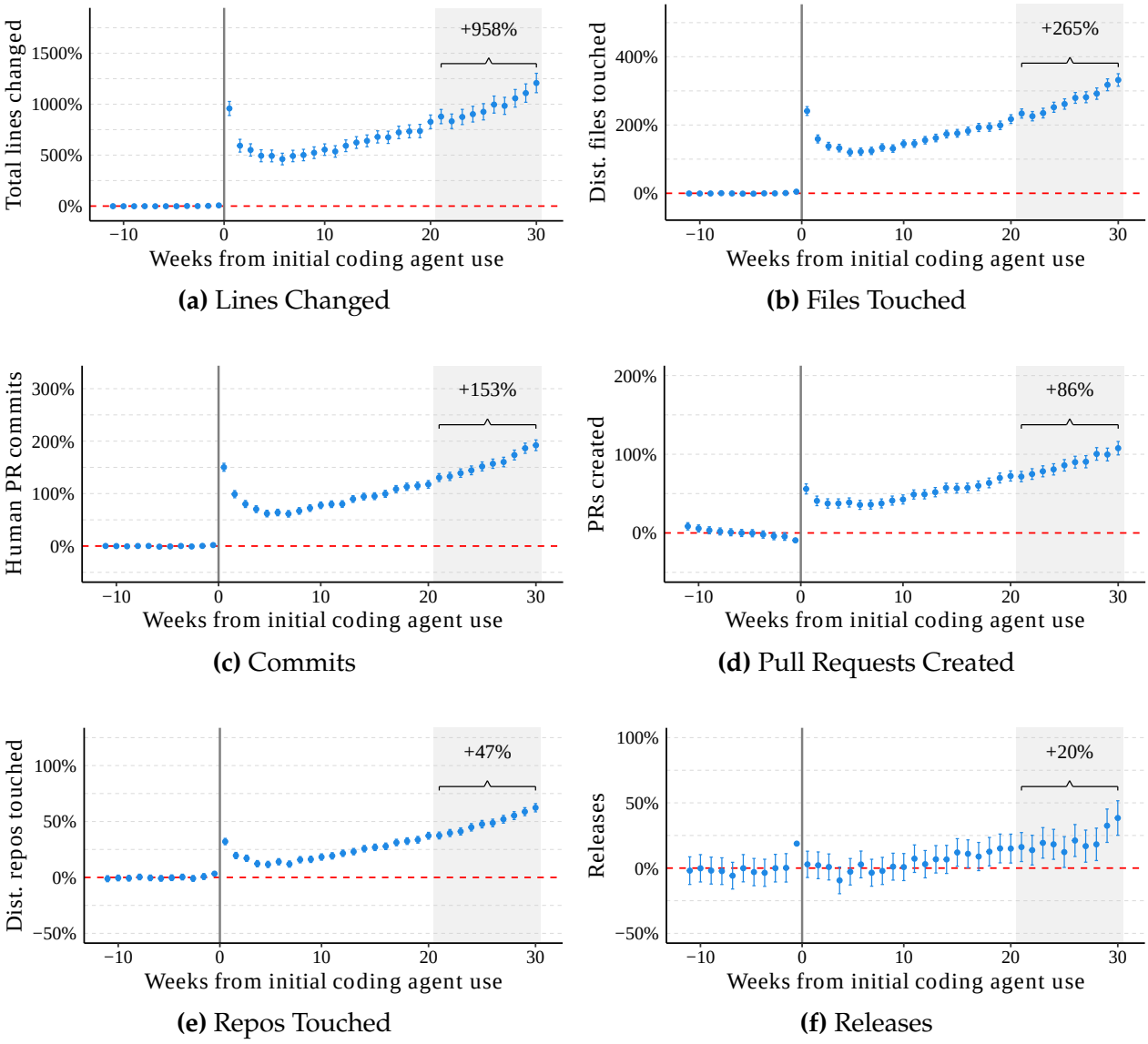
Notes: This table reports event-study coefficients (1% winsorized, weeks 21–30 bin, normalized by pre-period mean) measuring the effect of adoption of various generative AI coding tools on outcomes spanning the production hierarchy. Standard errors, clustered at the developer level (developer-by-tool for the pooled rows), in parentheses. Lines Chg = weekly total lines changed, Files Touched = distinct files touched, Commits = weekly commits, PRs = weekly PRs created, Repos Touched = distinct repositories contributed to, Releases = weekly repo release count. Releases are not reported for async tools because releases are measured at the repository level and cannot be split into agent- and human-authored components. The "GitHub Sync" row pools adopters identified from Microsoft usage data with GitHub Agent adopters, whose human-authored commits also reflect sync use (Section 4.2).

falling to 47.3% for repositories and 19.8% for releases. The decay across layers looks similar when we examine individual sync tools rather than the pooled effect.

For async agents, we do not report a release effect: releases are measured at the repository level and cannot be split into agent- and human-authored components. Across the remaining outcomes, the effects again decline across layers with one exception: the effect on pull requests is 68.5%, slightly larger than the 60.7% effect on commits. This pattern is consistent with async agents being particularly effective at the PR layer: they create pull requests autonomously, directly raising productivity at that layer rather than merely passing upstream gains through it.

Indeed, the model does not necessarily imply a strictly decreasing pattern for sync and async agents, since they enter at multiple layers: if they are more effective at higher layers, the effect size can in principle increase rather than decrease across layers. Either way, with multiple entry points, the model predicts dampened attenuation relative to a shock that enters only at the code-writing layer, as autocomplete does. We revisit this in Section 6.3 by mapping the empirical results directly to the model.

Figure 7: Productivity Effects of Sync Agents Across the Production Hierarchy



(a) Lines Changed **(b)** Files Touched

(c) Commits **(d)** Pull Requests Created

(e) Repos Touched **(f)** Releases

Notes: This figure presents matched event-study estimates of the sync effect across production layers, corresponding to the columns of Table 6. Outcomes are normalized by each developer's pre-period mean. The first post-adoption week is omitted as transitory effects distort the scale; the omitted coefficients are 1665.0%, 393.8%, 235.5%, −29.3%, 16.2%, and 13.4% for Panels 7(a)–7(f), respectively.

Figure 7 presents the corresponding event studies for the sync effect across all outcomes; the analogous per-tool event studies across all outcomes are reported in Figures OA-5–OA-10 in the Appendix. Several features are worth noting. First, pre-trends are flat for all outcomes, supporting the parallel trends assumption. Second, the effects on lines of code, files, and commits are increasing from roughly week 5 onward, consistent with the patterns in the main commit-level event studies and the opposite of the short-lived activity bias pattern of the placebo event studies. Third, the result for releases is the least visually

sharp: rather than a clear discontinuity at adoption, the effect builds gradually, which is consistent with the lag between writing code and shipping a release.

How do these magnitudes compare with other evidence? Our estimates are larger than those from studies of firm-level adoption: against our pooled sync effects of 153% on commits and 958% on lines of code, Chen and Stratton (2026) and Sarkar (2026) find that agent adoption raises commits or merges by 20% and 39%, and lines of code by 30% and 49%, respectively.[25] Smaller firm-level effects likely reflect organizational inertia: realizing the gains from AI tools requires complementary investments that take time (Bresnahan et al., 2002; Brynjolfsson et al., 2021), whereas the open-source developers we study work individually or in small teams and face less of this friction. By contrast, our estimates are closer to the rapidly growing gains that the frontier AI labs—organizations that have plausibly already made these complementary investments—report for their own engineers. For example, OpenAI reports that lines of code changed per developer rose roughly 600% between 2024 and mid-2026, and Anthropic reports that its engineers merged about 700% more code per day in the second quarter of 2026 than in 2024. Meta's reported numbers even mirror our attenuation pattern: internal code changes were up 220% year-over-year, while changes reaching users through new or upgraded features were up only 36%.[26]

To understand the robustness of the attenuation results, we use the placebo analysis and the alternative specifications described in Section 5.5. The placebo analysis (Table OA-10) implements the same empirical specification for non-AI coding tools and finds no attenuation. Under alternative specifications (level regressions, estimation on a common sample of developers active in every outcome, and alternative winsorization), the gradient's magnitude shifts somewhat but the attenuation is always present (Appendix Tables OA-7–OA-9). We further rule out code verbosity as an explanation for the attenuation: agents can write more verbose code, inflating counts of lines and files without a commensurate rise in substantive output. We examine verbosity empirically by looking at the share of comment lines in releases—the clearest symptom of AI verbosity—and find no upward trend between 2021 and 2026 (Table 7, Panel B). Moreover, verbosity cannot inflate counts of repositories or releases, yet we observe attenuation for these as well.

6.2 Mechanisms Underlying Attenuation

The previous section documented substantial attenuation of AI's productivity effects along the production hierarchy. We now examine two mechanisms that can generate this pattern:

[25] Timing may explain part of the gap: their samples end in March 2026 and January 2026, respectively, while ours runs later, and our evidence suggests that the effects are growing rapidly.

[26] OpenAI, "Research acceleration: The view inside OpenAI," September 6, 2026; Anthropic, "When AI builds itself," May 2026; Katie Paul, "Mark Zuckerberg had a bold plan to replace Meta staff with AI. Here's how it imploded," *Reuters*, August 26, 2026.

experimentation, whereby AI induces additional activity at the lower layers that never becomes shipped output, and *scope*, whereby the gains do reach the higher layers but accrue through the increased scope of each unit of output rather than in their number.

Which mechanism drives the attenuation matters for how we interpret it in light of our model. If the attenuation arises through scope—more inputs are generated at each layer and are packed into units of greater scope at the later stages, for example releases embodying more features—then the productivity gains are partially realized on the intensive margin, and the raw counts we use as outcomes do not fully reflect actual output. In this case, part of the measured attenuation is an artifact of measurement rather than evidence of bottlenecks.

The experimentation channel, by contrast, maps directly into the model and its weak-link hypothesis. If the attenuation comes from experimentation, it reflects AI expanding what developers generate at the earlier stages, while human review, quality control, and release capacity limit what reaches the final product. This can happen at several stages of production: developers may generate multiple versions of a piece of code in several iterations but later delete or abandon some of them; open branches to explore features that never make it into the final product; or create repositories to explore ideas they never ship.

Abandonment does not necessarily mean that the discarded work was of low quality: under complementarities, developers simply lack the bandwidth to process every candidate at the later stages, where human input is essential and AI is less effective, so they may simply be choosing among equally good alternatives based on taste, or keeping the best draw from a distribution. These abandoned coding activities are still an input to the production function, since the final output relies on generating and selecting among them, and our event-study measures include them: commits, branches, and repositories are counted whether or not the work ships. The experimentation implication of complementarities is consistent with the broader narrative of how software development is changing, especially the growing use of agents for rapid prototyping and iteration (Google, 2025).

We now turn from describing the two channels to measuring them. To gauge the role of scope, we examine whether the units at the higher layers have grown larger: we repeat the event study of Section 5.2 using lines per commit and files per commit as outcomes, and we also directly examine end-to-end attenuation by counting the number of lines of code contained in releases over time.

We examine evidence of experimentation at three levels of the hierarchy: (i) iteration at the code level, such as rewriting existing code, which indicates experimentation at the level of small tasks; (ii) branches that are created and deleted, which reflect experimentation at the feature level; and (iii) projects that are abandoned, which reflect experimentation

Table 7: Mechanisms of Attenuation: Experimentation vs. Scope

	Panel A. Event-Study Estimates					
	Lines per Commit	Files per Commit	Lines Added	Lines Deleted	Branches Created	Branches Deleted
Autocomplete	32.7	6.5	155.7	244.7	—	—
	(12.1)	(2.1)	(12.9)	(19.1)		
Pooled Sync	67.3	22.2	599.5	1115.7	78.6	109.0
	(7.9)	(1.3)	(12.3)	(20.5)	(2.4)	(3.6)

	Panel B. Aggregate Trends by Year					
	2021	2022	2023	2024	2025	2026
Share of repos abandoned (%)	60.0	62.4	62.7	63.7	67.6	69.0
	(0.4)	(0.4)	(0.3)	(0.4)	(0.7)	(0.7)
Median lines of code per release (000s)	5.21	5.49	5.63	6.94	9.03	13.42
	(0.06)	(0.06)	(0.05)	(0.06)	(0.07)	(0.13)
Median files per release	48	50	52	60	67	76
	(0.56)	(0.50)	(0.52)	(0.58)	(0.53)	(0.76)
Share of comment lines (%)	18.6	20.9	23.0	22.6	21.2	17.4
	(0.2)	(0.3)	(0.3)	(0.2)	(0.2)	(0.4)

Notes: Panel A reports matched event-study estimates of the effect of adopting each tool on the corresponding outcome (weeks 21–30 after adoption), normalized by each developer's pre-period mean; standard errors clustered at the developer level (developer-by-tool for the pooled sync rows) in parentheses. Branch outcomes are not available for autocomplete (—) because the branch data start in 2023. Panel B reports yearly statistics computed from the raw GitHub data for the repositories of the treated (adopter) sample, with standard errors in parentheses; the abandoned share is the share of repositories created in a given year that receive no activity after their first month (2026 covers repositories created in January–March, each observed for at least three months after its first month). Release size and comment statistics are computed from a random sample of 916,071 stable releases of repositories in our treated sample universe (code files only); medians with bootstrap standard errors, comment share pooled across releases; 2026 covers January–March.

at the level of ideas—starting a project to try something out and later abandoning it. For code-level iteration, we implement our event study with lines of code added and lines deleted separately as outcomes: an increase in deletions would indicate that the rise in gross lines translates into a smaller rise in net lines. For experimentation at the branch level, we examine the number of branches created and deleted and compare them with the other outcomes. For abandoned projects, we report the share of repositories that receive no activity after their first month, by creation cohort. Table 7 presents the results, separately for autocomplete and for pooled sync agents.

Starting with scope, the first two columns of Panel A in Table 7 report the effects on lines and files per commit: lines per commit rise by 33% and files per commit by 6% for autocomplete, and by 67% and 22% for sync agents. The end-to-end measure—the amount of code contained in released software, shown in Panel B—increases only by a factor of

about 2.6, from 5.21 to 13.42 thousand median lines per release, and the median number of files per release rises even less, by a factor of about 1.6 (from 48 to 76). Because these statistics are computed on the repositories of the adopters themselves, this modest growth is not an artifact of incomplete adoption: the comparison is not diluted by repositories of developers who never adopted AI tools. Thus, while the units at the higher layers have become larger, the magnitude of this increase is nowhere near what would be required to account for the attenuation pattern we document.

Turning to experimentation, the third and fourth columns of Panel A show that deletions increase by a factor of 12.2, whereas additions increase only by a factor of 7.0, suggesting a substantial rise in the reworking of code. The last two columns of Panel A report a 79% increase in the number of branches created, providing direct evidence of experimentation at the feature level, since branches are most likely to be created when a developer tries something new. Finally, the first row of Panel B shows that the share of repositories that are abandoned rises by 9 percentage points between the 2021 and 2026 cohorts. The rise accelerates with the arrival of agentic tools: the share increases by about 1.2 percentage points per year over 2021–2024, and by 3.9 percentage points in 2025 alone.[27]

Taken together, these results suggest that increased scope explains only a small fraction of the attenuation; most of it instead reflects experimentation at multiple layers of the production hierarchy. AI enables developers to try far more, while human bottlenecks at the later stages filter out most of these attempts.

6.3 Quantifying Complementarities in Software Production

We have established that productivity gains attenuate across layers, and we have provided suggestive evidence that this attenuation reflects complementarities and bottlenecks along the production chain. We now turn to quantifying these complementarities using the production model of Section 3. We showed there that the elasticity of substitution between upstream and effective input is the key parameter governing the shape of attenuation across layers. We can therefore invert this relationship, using our estimated attenuation pattern to back out the structural parameters that govern how task-level productivity translates into final output. We note that our production function is highly stylized, and we do not interpret this exercise as identification of the actual production function. However, mapping the data into these parameters provides suggestive evidence on the potential complementarities.

To do this, we rely on Proposition 1, which showed that pass-through at each layer is

[27] To avoid censoring at the right end of the sample, the 2026 cohort includes only repositories created in the first three months of the year, leaving a buffer of at least three months after each repository's first month closes to detect abandonment.

governed by two parameters: θ, the output elasticity of each layer with respect to upstream input, and σ, the elasticity of substitution between upstream output and effective input. Our goal is to find values of (θ, σ) that, applied to the model, reproduce the empirical attenuation pattern.

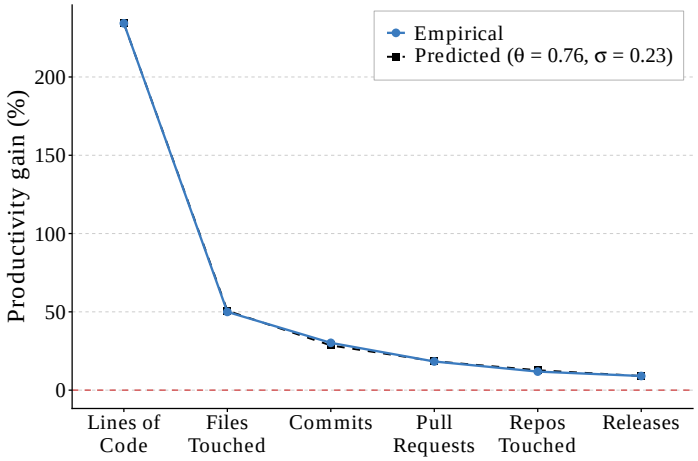
We perform this calibration using the autocomplete estimates alone. Autocomplete enters the hierarchy only at the code-writing layer, so the model's prediction is simple: we take the empirical lines-of-code effect as the layer-1 shock and apply the pass-through formula recursively to obtain predicted gains at each higher layer. Sync and async agents, in contrast, intervene at multiple layers (Section 3), and we cannot separately identify their layer-specific productivity inputs from the attenuation pattern alone. For tractability we assume θ and σ are constant across layers, leaving two free parameters. We then choose (θ, σ) to minimize the sum of squared log differences between predicted and empirical gains at all six layers. Appendix D.8 provides the formal details of the calibration exercise.

How are these two parameters identified from the data? They affect the attenuation pattern in qualitatively different ways, and both signatures are visible empirically. The parameter θ shifts the overall rate of decay: a higher θ means upstream input has a larger output elasticity, so the same upstream gain propagates more strongly through each layer. The parameter σ shapes the decay through the share of the log gain that survives from one layer to the next. When $\sigma < 1$ (complements), this share rises across layers: the layers just above the shock absorb a disproportionate share of the gain, and what survives propagates more easily thereafter. When $\sigma = 1$, it is constant at θ. When $\sigma > 1$ (substitutes), it falls across layers (Proposition 2). The overall steepness of the empirical decay therefore identifies θ, and the way this surviving share moves across layers identifies σ.

Figure 8 shows the empirical autocomplete attenuation pattern and the model prediction at the best-fit parameters, $\theta \approx 0.76$ and $\sigma \approx 0.23$. The estimated elasticity of substitution is well below one: upstream output and effective input are strong complements, combined in roughly fixed proportions. This is why the attenuation we observe is steep and front-loaded. The estimate of θ implies that at each layer output has an elasticity of roughly 0.76 with respect to upstream input and roughly 0.24 with respect to the layer's own effective input.

We next construct a counterfactual prediction for sync and async agents under the assumption that these tools affect only lines of code (layer 1). For each tool, we take its empirical lines-of-code effect as the layer-1 shock and apply the recursive pass-through formula with the (θ, σ) estimated from autocomplete. Figure 9 compares these counterfactual predictions with the empirical effects. For both sync and async agents, the empirical

Figure 8: Calibrated Model Fit for Autocomplete



Notes: This figure shows the calibrated model's fit to the autocomplete attenuation pattern ($\theta = 0.76$, $\sigma = 0.23$). Blue circles show the empirical effects at each layer; black squares show the model prediction.

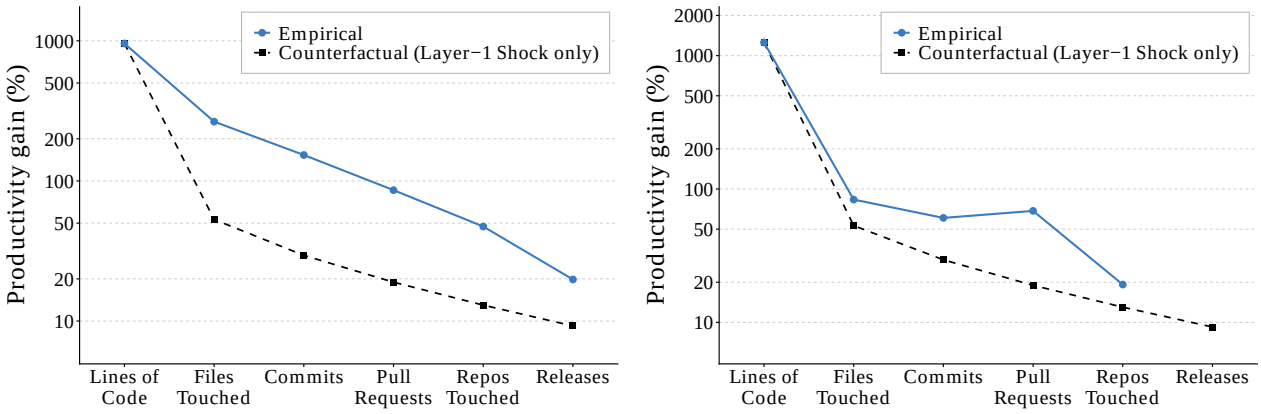
effects lie above the predicted gains at every intermediate layer. These deviations are consistent with sync and async agents intervening directly at higher layers of the production hierarchy, not merely passing through their layer-1 effect.

To further understand the role of complementarities, we repeat the calibration for the two alternative models of attenuation developed in Section 3.5 and Appendix B, using the same procedure: a Cobb–Douglas production function, which has no complementarities, and a CES specification with decreasing returns to scale. Both alternatives fail, for different reasons: the Cobb–Douglas loss is more than 100 times the baseline's, while the decreasing-returns specification matches the pattern only with economically implausible parameter values (Appendix Table OA-1). We therefore conclude that complementarities are the primary channel behind the attenuation pattern.

We also ask whether developers reallocate their inputs across layers in response to the productivity shock, one of the extensions of the model. The data provide no support for full reoptimization: allowing reallocation does not improve the fit. This suggests that the reallocation margin does not operate in the short run and takes time, consistent with J-curve dynamics, in which the gains from a new technology materialize only after complementary adjustments (Brynjolfsson et al., 2021). Such a delay is natural in our setting: we study the period immediately after these tools became available, before developers could plausibly have completed such adjustments.

The results in this section show that productivity effects attenuate from lines of code toward final output, in a pattern consistent with complementarities and driven by experimentation. This provides direct empirical evidence for the weak-link hypothesis widely

Figure 9: Measuring the Layers where Sync and Async Agents Intervene



(a) Sync Agents **(b)** Async Agents

Notes: Each panel uses the autocomplete-calibrated $(\theta, \sigma) = (0.76, 0.23)$ to predict the attenuation pattern for sync (Panel a) and async (Panel b) agents, taking each tool's empirical lines-of-code effect as the layer-1 shock. Blue circles show the empirical effects at each layer; black squares show the model predictions.

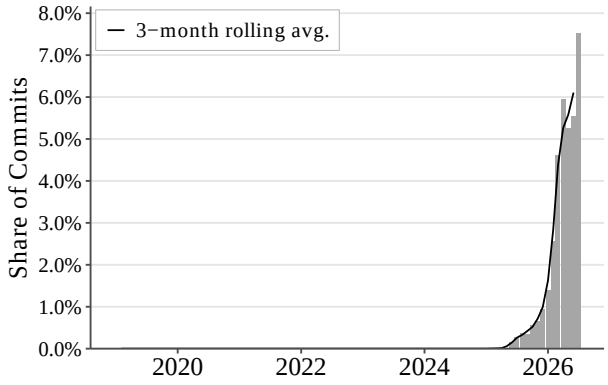
discussed in the macroeconomic productivity literature (Jones, 2011; Aghion et al., 2019; Jones, 2026). At the same time, the bottleneck is moving up the production chain as AI capabilities advance—autocomplete intervenes only at the code-writing layer, while sync and async agents intervene at progressively higher layers—so the gap between task-level effects and effects on shipped software may narrow over time as tools automate higher layers of the hierarchy.

7 Aggregate Evidence from GitHub and Software Marketplaces

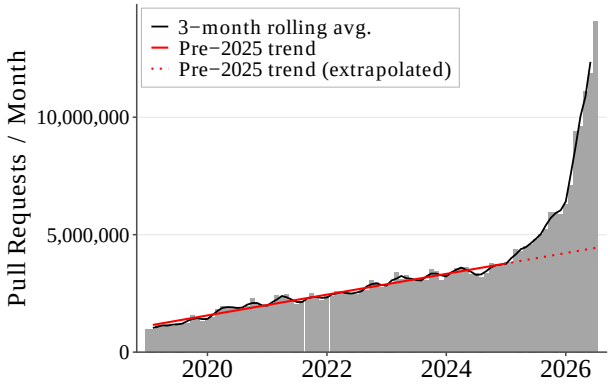
Our developer-level results document substantial productivity gains from AI coding tools, with effects that attenuate sharply across the production hierarchy. A natural question is whether these micro-level effects translate into aggregate changes in software production. This question echoes a long-standing debate in economics. In 1987, Robert Solow famously observed that "you can see the computer age everywhere but in the productivity statistics" (Solow, 1987). Decades later, the same concern has resurfaced with AI. While individual studies document large task-level productivity gains (e.g., Peng et al., 2023; Noy and Zhang, 2023; Brynjolfsson et al., 2025; Cui et al., 2026), whether these gains will reach aggregate output remains an open question (Acemoglu, 2025; Jones, 2026; Karger et al., 2026).

We examine this question in two parts. First, we study aggregate coding activity on GitHub. This analysis provides a direct aggregate counterpart to our developer-level event studies. Second, we turn to the output of software on major distribution marketplaces, examining whether the productivity gains we document translate into more apps being created and, ultimately, used by end users.

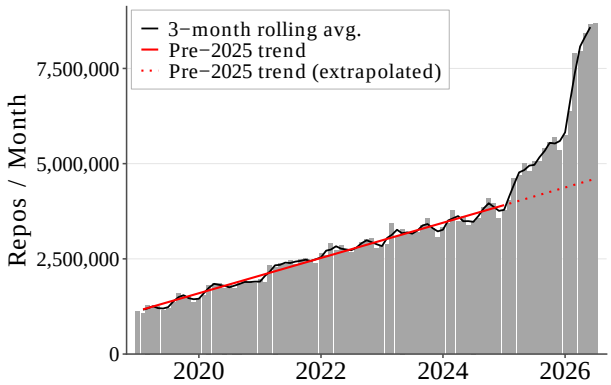
Figure 10: Aggregate Coding Activity on GitHub



(a) Claude Code Commit Share



(b) Pull Requests Merged



(c) New Repos Created

Notes: This figure plots monthly aggregate activity on public GitHub repositories from 2019 through mid-2026. Bars show monthly counts; the solid line is a rolling average. Panel 10(a) plots the share of public commits signed or authored by Claude Code: the numerator is a count of Claude-signed commits from claudescode.dev, and the denominator is the GitHub Search API's count of total public commits, which is an estimate from a sampled index rather than an exact total, so the share is approximate; developers may also commit work done by Claude themselves, so this is likely an undercount. Panels 10(b) and 10(c) are both drawn from the GitHub Search API, and in each the red line is a linear trend fit through 2024, extrapolated (dotted) thereafter; pull requests merged counts all merged PRs on public repositories, and new repos counts non-fork public repositories created. Appendix C.6 gives full details on all three series.

7.1 Aggregate Coding Activity on GitHub

We begin by establishing that aggregate adoption of AI coding tools on GitHub is large enough to make our developer-level effects potentially detectable in platform-wide totals. Figure 10(a) plots the share of commits on public repositories that are signed or authored by just one agentic coding tool: Claude Code. The share has risen rapidly since the tool's release in early 2025 and by mid-2026 exceeds 7% of all public commits. Since developers frequently commit AI-generated work under their own name and use many tools beyond Claude Code, this is almost certainly an undercount of total AI-assisted coding activity.

Figures 10(b) and 10(c) turn to aggregate platform activity directly, plotting monthly

counts of merged pull requests and newly created (non-fork) public repositories from 2019 through mid-2026. Both series grow roughly linearly from 2019 through 2024, in line with steady organic platform growth. Around early 2025, this trend breaks: both series accelerate sharply relative to the pre-2025 trend. The acceleration in merged pull requests is also visibly larger than the acceleration in new repositories created, consistent with the attenuation pattern documented in Section 6.[28]

7.2 New Apps and Usage in Software Marketplaces

In contrast to aggregate productivity in the broader economy, software output is reasonably observable: software marketplaces record when new apps are published and how widely they are used. We use this transparency to ask whether the developer-level productivity gains documented above translate into measurable changes in software output and software consumption.

7.2.1 Data and Measurement

We assemble monthly panels for four of the largest software marketplaces: the Apple App Store (iOS), the Google Play Store (Android), the Chrome Web Store (browser extensions), and SourceForge (a legacy platform for desktop software). Together these four span different developer populations, from mobile consumer apps to browser extensions and desktop software.

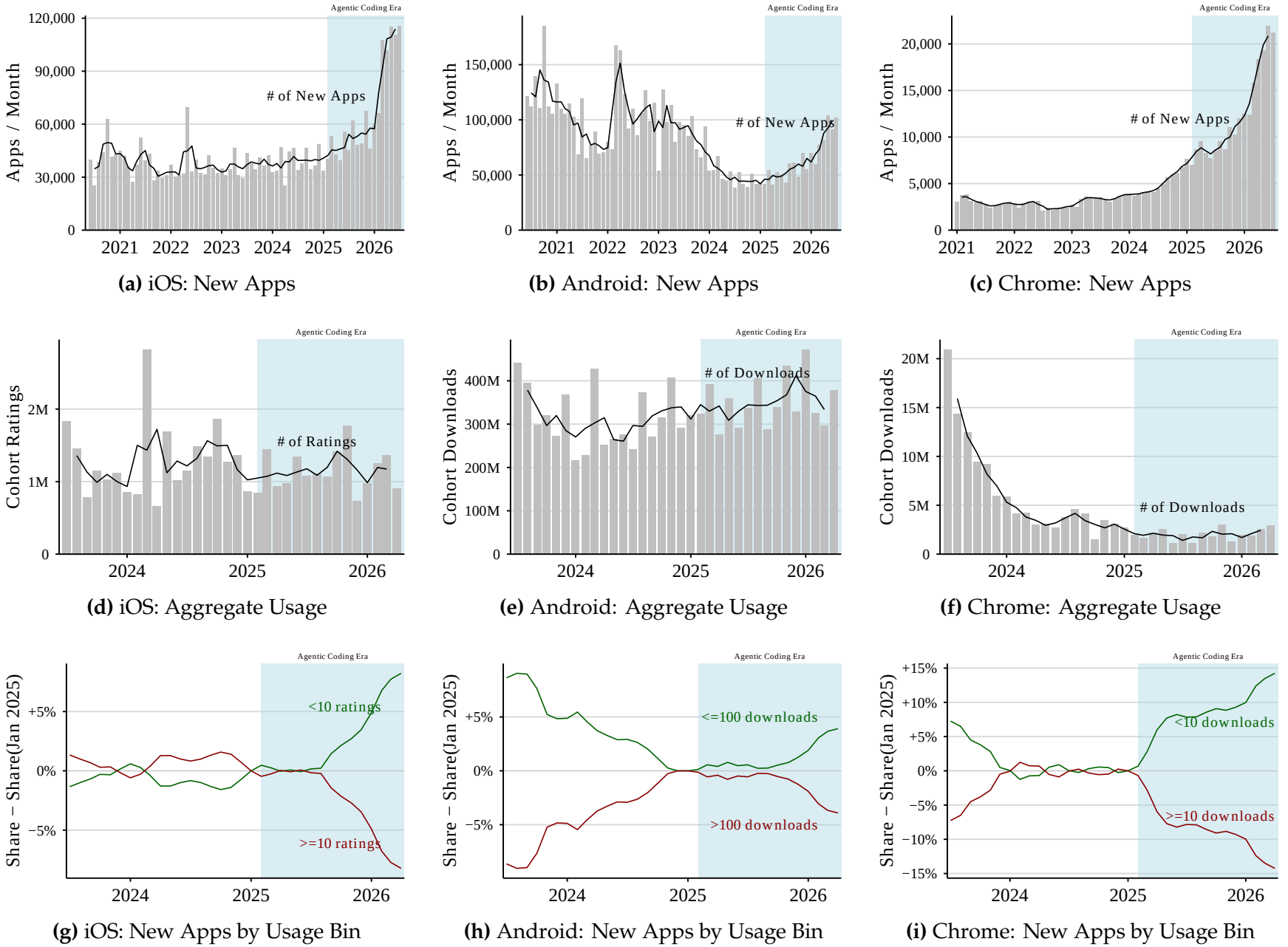
For each marketplace we observe each app's first-seen date and cumulative end-user usage thereafter. The usage measure is the rating count for iOS (where we do not directly observe installs) and the cumulative install count for the other marketplaces. Using these data, we construct monthly cohorts, each consisting of the non-incumbent apps that first appeared in that month, and we track the usage they attract within their first three months. This window is short enough to include cohorts near the end of our sample and long enough that the usage signal is meaningful. Our results are robust to other windows.

7.2.2 Releases of New Applications

The top row of Figure 11 plots monthly counts of new releases for each marketplace. To help the reader assess whether agentic coding accelerated new releases, we highlight the

[28] GitHub itself has pointed to this surge: in a postmortem of a major outage on August 17, 2026, GitHub's CTO reported that monthly commits had more than doubled since April 2026, from 1.4 to 2.9 billion, and cited this growth as the source of the capacity strain behind the outage. See GitHub Blog—The August 17 outage and the work ahead.

Figure 11: New Apps and Their Usage Across Software Marketplaces



(a) iOS: New Apps **(b)** Android: New Apps **(c)** Chrome: New Apps

(d) iOS: Aggregate Usage **(e)** Android: Aggregate Usage **(f)** Chrome: Aggregate Usage

(g) iOS: New Apps by Usage Bin **(h)** Android: New Apps by Usage Bin **(i)** Chrome: New Apps by Usage Bin

Notes: Monthly activity across the iOS, Android, and Chrome app stores. Top row: counts of newly released apps, with a 3-month centered rolling average. Middle row: total user-side activity (ratings for iOS; downloads for Android and Chrome) accumulated in the 3 months after entry, summed across each monthly cohort. Bottom row: cohort share in a low-usage vs. high-usage bucket (rating count for iOS; download count for Android and Chrome), level-shifted to zero in January 2025. Shaded region: agentic-coding era (February 2025 onward).

period from February 2025 onward as the agentic-coding era on each plot.[29]

All three platforms show either an acceleration in entry or a break from a prior decline, aligned broadly in time with the rise of agentic coding tools. The Apple App Store shows the clearest acceleration. New iOS apps run between roughly 33,000 and 45,000 per month from 2023 through early 2025, then rise sharply over the course of 2025, reaching approximately 108,000 per month by April 2026. The acceleration further gathers pace in early 2026, consistent with the Claude Code effect rising over time as the underlying model improves (Figure 5(b)).

The Google Play Store shows a different pattern. Monthly new releases had declined steadily from a 2020 peak of about 145,000 to about 42,000 in January 2025, and the trajectory then breaks: releases stabilize and recover through the end of our sample, reaching about 99,000 per month by mid-2026. The Chrome Web Store shows a slower but visible acceleration that predates the agentic coding era, with new extensions rising roughly 7-fold since 2023, though growth picks up further with the arrival of coding agents in 2025.[30]

7.2.3 Consumption of New Apps

We now investigate whether the increase in new releases translates into additional consumption, which is informative about, though not a direct measure of, consumer surplus. We first examine total cohort-level usage. The middle row of Figure 11 plots, for each entry month, the sum of usage over the next three months across all apps in that cohort. If the additional apps had generated meaningful consumption, the aggregate would rise with the size of the cohort. Instead, aggregate usage fails to keep pace in all three marketplaces: iOS cohort ratings are roughly stable between 2024 and 2025, Android cohort downloads rise only modestly (far less than the number of new releases), and Chrome cohort downloads fall rapidly, though they had already been falling in the pre-period.

The fact that aggregate usage does not rise with entry rules out large increases in consumer surplus from the introduction of new blockbuster apps. It also rules out a "long-tail" surplus channel, in which a large number of newly released niche apps each attract a small audience but, integrated across the cohort, generate substantial total consumer surplus (Brynjolfsson et al., 2003). Under the long-tail story, each app's user base would be small but the aggregate would rise with the number of new releases. While this channel

[29] We choose this date because GitHub launched Copilot Agent Mode, among the first widely used agentic coding products, in late February 2025; other agentic tools followed soon after. See the announcement from Visual Studio Code.

[30] We report SourceForge in Appendix Figure OA-17 because its download counts are too small to support a meaningful analysis of usage; we therefore show only the number of new projects, which increases slightly.

has been documented for AI-written books on Amazon (Reimers and Waldfogel, 2026), it is not visible in the software marketplaces we study.

Still, flat aggregate usage does not by itself imply that consumer surplus is unaffected, as the introduction of additional apps could have improved *matching* between consumers and apps. From a consumer's perspective, apps are partial substitutes, since time spent on any one app is bounded by total leisure time, so an additional new app need not raise total usage. The new apps could still raise consumer surplus by letting each consumer choose the variant best suited to her preferences, with the gains showing up as better matches rather than more usage.

To assess the matching channel, we examine the share of each monthly cohort that fails to reach a moderate audience within three months. We classify an app as failing to reach an audience if it accumulates fewer than 10 ratings (iOS), fewer than 10 downloads (Chrome), or at most 100 downloads (Android) in its first three months.[31] The bottom row of Figure 11 plots the time path of this share (in green) and its complement (in dark red) for each marketplace, level-shifted to zero in January 2025. In all three marketplaces the share of releases that fail to reach an audience trends upward in the agentic era: between January 2025 and April 2026, it rises from about 78% to 87% on iOS, from 19% to 33% on Chrome, and from about 22% to 26% on Android. If the additional apps were better-matched substitutes, users switching to them would generate at least the modest demand needed to clear the threshold; instead, the marginal apps are increasingly invisible to end users. This argues against the matching channel as well.

7.2.4 Discussion

The marketplace evidence has two parts. New app releases have accelerated, while cohort-level usage has not. The first finding is consistent with the hierarchy model of Section 3: entry has increased across marketplaces, but by less than the task-level productivity estimates would imply. These increases nevertheless indicate that the rise in coding activity we document on GitHub has reached the marketplaces.[32]

The second pattern, that usage fails to keep pace with entry, admits multiple explanations. One reading is supply-side: AI relaxes the constraints on producing and publishing software, so more projects reach publication. But publication need not be the binding layer

[31] We use platform-specific thresholds because usage intensity differs substantially across marketplaces; a uniform cutoff would push the small-audience share close to zero or one in some marketplaces, leaving little variation to study.

[32] The effects on the number of new apps in the marketplaces are larger than the developer-level release event-study estimates. This is partly because GitHub is observed earlier than the software marketplaces: the marketplace data end in August 2026, whereas the GitHub estimation panel ends in mid-April 2026 (Appendix D.3), and the 30-week post-adoption windows fall even earlier for early adopters.

for market-valued output: developers must still test, polish, and iterate toward market fit, and these higher-level tasks may remain constrained even when AI lowers the cost of writing code. The marketplace evidence would then be the attenuation pattern writ large: gains at upstream layers raise the flow of releases, but usage-weighted output rises much less.

An alternative explanation, which we cannot rule out, is demand-side congestion. Even if the quality of new apps were unchanged, user attention and discovery channels (search rankings, recommendations, word of mouth) are scarce and need not expand with the number of entrants. As more apps compete for a fixed pool of attention, average per-app usage falls. This is not a production bottleneck, but it looks the same in the data: more publications without proportionally more usage.[33]

8 Conclusion

The capabilities of generative AI tools for software development have advanced rapidly, moving from autocomplete to fully autonomous coding agents in just a few years. Do these capability gains translate into higher real-world *task-level productivity*? And even if they do, do task-level gains translate into more *final output*? These questions are central to understanding AI's economic impact. In developer-level matched event studies, we find that successive generations of AI coding tools produce increasingly large task-level productivity effects: autocomplete raises weekly commits by roughly 30%, sync agents raise the cumulative effect to roughly 180%, and async agents to roughly 240%. Yet these gains attenuate sharply across the production hierarchy: sync agents lead to a 958% increase in lines of code and an 86% increase in pull requests, but releases rise by only 20%. Aggregate coding activity on GitHub mirrors this pattern.

To interpret this attenuation, we introduce a hierarchical model of software production in which output is produced through the sequential aggregation of tasks. The model highlights the elasticity of substitution between upstream output and downstream human effort as the key parameter governing how task-level gains propagate to final output. Calibrating the model to our estimates yields an elasticity of 0.23, pointing to strong complementarity between AI and human inputs along the production chain. Direct evidence on mechanisms supports this reading: the attenuation reflects experimentation that is fil-

[33]Several caveats about our analysis are worth noting. First, our sample runs only through mid-2026, leaving under a year and a half of post-launch data for agentic tools; the usage response may simply be slow, since discovery and adoption take time and supply-side learning from user feedback requires multiple product iterations. Second, we cannot fully separate productivity-driven entry from demand-driven entry; some new apps may have been written specifically to serve AI-related demand. Third, our demand measures are imperfect proxies for consumer surplus: ratings and downloads need not move one-for-one with the value an app generates for its users, particularly for free apps.

tered out at human-bottlenecked stages (more reworking of code, more branches created, more projects abandoned), rather than larger units of output.

The same logic motivates our analysis of software marketplaces: if AI increases the supply of software, does that software reach users, and is it actually used? Across all four marketplaces, new app creation has accelerated since early 2025, but total usage has not kept pace, and the share of new apps that fail to reach even a modest audience has increased. This pattern admits two interpretations: either higher production layers (testing, polishing, iterating toward market fit) still bottleneck market-valued output, mirroring in aggregate the attenuation we document, or a consumer-side bottleneck (discovery and adoption) prevents increased supply from translating into usage.

Our evidence comes from open-source development, but the pattern extends to firms. Chen and Stratton (2026), using proprietary data on enterprise software development, find the same attenuation, with code review as the binding constraint. If anything, these bottlenecks should bind more tightly inside firms, where shipping code involves review and approval processes that open-source developers largely avoid, and easing these bottlenecks will require complementary organizational investments (Brynjolfsson et al., 2021).

Taken together, our results provide empirical evidence for a more general principle. When stages of production are complementary, automating one stage has bounded effects on final output: a vertical analog of the "weak links" logic in the macroeconomic growth literature. This offers one resolution of the tension between large task-level productivity estimates and muted aggregate projections: both can hold at once when the stages of production are strong complements. In software, the binding constraint appears to be shifting from writing code to reviewing, integrating, and distributing it. Whether future generations of AI tools can ease these downstream constraints—by producing higher-quality code that requires less human review, automating review and integration, or improving discovery and adoption—will determine whether the large task-level productivity gains we document ultimately translate into commensurate increases in shipped and used software.

References

Acemoglu, D. (2025). The Simple Macroeconomics of AI. *Economic Policy 40*(121), 13–58.

Acemoglu, D. and D. Autor (2011). Skills, Tasks and Technologies: Implications for Employment and Earnings. In *Handbook of Labor Economics*, Volume 4, pp. 1043–1171.

Acemoglu, D. and P. Restrepo (2018). The Race between Man and Machine: Implications of Technology for Growth, Factor Shares, and Employment. *American Economic Review 108*(6), 1488–1542.

Acemoglu, D. and P. Restrepo (2019). Automation and New Tasks: How Technology Displaces and Reinstates Labor. *Journal of Economic Perspectives 33*(2), 3–30.

Acemoglu, D. and P. Restrepo (2022). Tasks, Automation, and the Rise in U.S. Wage Inequality. *Econometrica 90*(5), 1973–2016.

Aghion, P., B. F. Jones, and C. I. Jones (2019). Artificial Intelligence and Economic Growth. In A. Agrawal, J. Gans, and A. Goldfarb (Eds.), *The Economics of Artificial Intelligence: An Agenda*, pp. 237–290. University of Chicago Press.

Agrawal, A. K., J. McHale, and A. Oettl (2026). Enhancing Worker Productivity Without Automating Tasks: A Different Approach to AI and the Task-Based Model. *NBER Working Paper, No. 34781*.

Autor, D. and N. Thompson (2025). Expertise. *NBER Working Paper, No. 33941*.

Autor, D. H., F. Levy, and R. J. Murnane (2003). The Skill Content of Recent Technological Change: An Empirical Exploration. *Quarterly Journal of Economics 118*(4), 1279–1333.

Baqaee, D. R. and E. Farhi (2019). The Macroeconomic Impact of Microeconomic Shocks: Beyond Hulten's Theorem. *Econometrica 87*(4), 1155–1203.

Becker, J., N. Rush, E. Barnes, and D. Rein (2025). Measuring the Impact of Early-2025 AI on Experienced Open-Source Developer Productivity. *arXiv Preprint arXiv:2507.09089*.

Benzell, S. G. and K. R. Myers (2026). Automation Experiments and Inequality. *NBER Working Paper, No. 34668*.

Bick, A., A. Blandin, and D. J. Deming (2025). The Rapid Adoption of Generative AI. *Management Science*.

Bloom, N., R. Sadun, and J. Van Reenen (2012). Americans Do IT Better: US Multinationals and the Productivity Miracle. *American Economic Review 102*(1), 167–201.

Bresnahan, T. F., E. Brynjolfsson, and L. M. Hitt (2002). Information Technology, Workplace Organization, and the Demand for Skilled Labor: Firm-Level Evidence. *The Quarterly Journal of Economics 117*(1), 339–376.

Bresnahan, T. F. and M. Trajtenberg (1995). General Purpose Technologies 'Engines of Growth'? *Journal of Econometrics 65*(1), 83–108.

Brynjolfsson, E., B. Chandar, and R. Chen (2026). Canaries in the Coal Mine? Six Facts about the Recent Employment Effects of Artificial Intelligence. Working paper.

Brynjolfsson, E. and L. M. Hitt (2000). Beyond Computation: Information Technology, Organizational Transformation and Business Performance. *Journal of Economic Perspectives 14*(4), 23–48.

Brynjolfsson, E., Y. J. Hu, and M. D. Smith (2003). Consumer Surplus in the Digital Economy: Estimating the Value of Increased Product Variety at Online Booksellers. *Management Science 49*(11), 1580–1596.

Brynjolfsson, E., D. Li, and L. Raymond (2025). Generative AI at Work. *The Quarterly Journal of Economics 140*(2), 889–942.

Brynjolfsson, E., D. Rock, and C. Syverson (2021). The Productivity J-Curve: How Intangibles Complement General Purpose Technologies. *American Economic Journal: Macroeconomics 13*(1), 333–372.

Chen, F. and J. Stratton (2026). Artificial Intelligence in the Firm: Bottlenecks in Software Production. Working paper.

Chen, V., A. Talwalkar, R. Brennan, and G. Neubig (2025). Code with Me or for Me? How Increasing AI Automation Transforms Developer Workflows. *arXiv Preprint arXiv:2507.08149*.

Choi, J. H. and D. Schwarcz (2025). AI Assistance in Legal Analysis: An Empirical Study. *Journal of Legal Education 73*(2).

Cui, Z., M. Demirer, S. Jaffe, L. Musolff, S. Peng, and T. Salz (2026). The Effects of Generative AI on High-Skilled Work: Evidence from Three Field Experiments with Software Developers. *Management Science*. Forthcoming.

Davidson, T., B. Halperin, T. Houlden, and A. Korinek (2026). When Does Automating AI Research Produce Explosive Growth? Feedback Loops in Innovation Networks. *NBER Working Paper, No. 35155*.

Dell'Acqua, F., E. McFowland III, E. R. Mollick, H. Lifshitz-Assaf, K. C. Kellogg, S. Rajendran, L. Krayer, F. Candelon, and K. R. Lakhani (2026). Navigating the Jagged Technological Frontier: Field Experimental Evidence of the Effects of Artificial Intelligence on Knowledge Worker Productivity and Quality. *Organization Science 37*(2), 403–423.

Demirer, M. (2026). Production Function Estimation with Factor-Augmenting Technology: An Application to Markups. *Econometrica*. Forthcoming.

Demirer, M., J. J. Horton, N. Immorlica, B. Lucier, and P. Shahidi (2026). Chaining Tasks, Redefining Work: A Theory of AI Automation. *NBER Working Paper, No. 34859*.

Dillon, E. W., S. Jaffe, N. Immorlica, and C. T. Stanton (2025). Shifting Work Patterns with Generative AI. *NBER Working Paper, No. 33795*.

DORA (2024). Accelerate State of DevOps Report 2024. *Google Cloud*. https://dora.dev/research/2024/dora-report/.

Eloundou, T., S. Manning, P. Mishkin, and D. Rock (2024). GPTs Are GPTs: Labor Market Impact Potential of LLMs. *Science 384*(6702), 1306–1308.

Felten, E. W., M. Raj, and R. Seamans (2023). Occupational Heterogeneity in Exposure to Generative AI. SSRN Working Paper 4414065.

Forsgren, N., M.-A. Storey, C. Maddila, T. Zimmermann, B. Houck, and J. Butler (2021). The SPACE of Developer Productivity. *Communications of the ACM 64*(6), 46–53.

Freund, L. B. and L. F. Mann (2026). Job Transformation, Specialization, and the Labor Market Effects of AI. *Working Paper*.

Fruits, E. and K. Stout (2026). AI, Productivity, and Labor Markets: A Review of the Empirical Evidence. *International Center for Law & Economics Issue Brief*.

Gans, J. S. and A. Goldfarb (2026). O-Ring Automation. *NBER Working Paper, No. 34639*.

Garicano, L. (2000). Hierarchies and the Organization of Knowledge in Production. *Journal of Political Economy 108*(5), 874–904.

Garicano, L., J. Li, and Y. Wu (2026). Weak Bundle, Strong Bundle: How AI Redraws Job Boundaries. *CEPR Discussion Paper, No. 21453*.

Goldberg, S. and H. T. Lam (2026). Generative AI & Creative Goods: Market Expansion, Crowd-Out, and Copyright. *SSRN Working Paper, No. 5152649*.

Google (2025). DORA: State of AI-assisted Software Development. Available at https://dora.dev/dora-report-2025/.

Gortmaker, J. (2026). Open Source Software Policy in Industry Equilibrium. Working paper.

Hampole, M., D. Papanikolaou, L. D. W. Schmidt, and B. Seegmiller (2025). Artificial Intelligence and the Labor Market. *NBER Working Paper, No. 33509*.

Handa, K., A. Tamkin, M. McCain, S. Huang, E. Durmus, S. Heck, J. Mueller, J. Hong, S. Ritchie, T. Belonax, K. K. Troy, D. Amodei, J. Kaplan, J. Clark, and D. Ganguli (2025). Which Economic Tasks are Performed with AI? Evidence from Millions of Claude Conversations. *arXiv Preprint arXiv:2503.04761*.

He, H., C. Miller, S. Agarwal, C. Kästner, and B. Vasilescu (2026). Speed at the Cost of Quality: How Cursor AI Increases Short-Term Velocity and Long-Term Complexity in Open-Source Projects. *Proceedings of the 23rd International Conference on Mining Software Repositories (MSR '26)*.

Hoffmann, M., S. Boysel, F. Nagle, S. Peng, and K. Xu (2024). Generative AI and the Nature of Work. *SSRN Working Paper, No. 5007084*.

Hoffmann, M., F. Nagle, and Y. Zhou (2024). The Value of Open Source Software. *SSRN Working Paper, No. 4693148*.

Hulten, C. R. (1978). Growth Accounting with Intermediate Inputs. *Review of Economic Studies 45*(3), 511–518.

Humlum, A. and E. Vestergaard (2025). Still Waters, Rapid Currents: Early Labor Market Transformation Under Generative AI. *NBER Working Paper, No. 33777*.

Ide, E. and E. Talamàs (2025). Artificial Intelligence in the Knowledge Economy. *Journal of Political Economy 133*(12), 3762–3800.

Jabarian, B. and L. Henkel (2026). Voice AI in Firms: A Natural Field Experiment on Automated Job Interviews. *SSRN Working Paper, No. 5395709*.

Johnston, D., D. Holtz, A. M. Richmond, C. Ong, P. Tambe, and A. Chatterji (2026). The Shift to Agentic AI: Evidence from Codex. arXiv Preprint arXiv:2606.26959.

Jones, C. I. (2011). Intermediate Goods and Weak Links in the Theory of Economic Development. *American Economic Journal: Macroeconomics 3*(2), 1–28.

Jones, C. I. (2026). AI and Our Economic Future. *Journal of Economic Perspectives 40*(3), 3–22.

Jones, C. I. and C. Tonetti (2026). Past Automation and Future A.I.: How Weak Links Tame the Growth Explosion. *Working Paper*.

Kalyani, A., N. Bloom, M. Carvalho, T. Hassan, J. Lerner, and A. Tahoun (2025). The Diffusion of New Technologies. *The Quarterly Journal of Economics 140*(2), 1299–1365.

Karger, E., O. Kuusela, J. Abaluck, K. A. Bryan, B. Halperin, T. R. Jones, C. Murphy, P. Trammell, M. Reynolds, D. Mayland, R. Viswanathan, A. Mittal, R. Ceppas de Castro, J. Rosenberg, and P. Tetlock (2026). Forecasting the Economic Effects of AI. *NBER Working Paper, No. 35046*.

Kim, H., D. Kim, and R. Koning (2026). Mapping AI into Production: A Field Experiment on Firm Performance. *SSRN Working Paper, No. 6513481*.

Kim, H. and R. Koning (2026). AI-Native Firms. *SSRN Working Paper, No. 6905079*.

Ko, A. J. (2019). Why We Should Not Measure Productivity. In C. Sadowski and T. Zimmermann (Eds.), *Rethinking Productivity in Software Engineering*, pp. 21–26. Apress.

Korinek, A. and D. Suh (2024). Scenarios for the Transition to AGI. *NBER Working Paper, No. 32255*.

Kreitmeir, D. and P. A. Raschky (2024). The Heterogeneous Productivity Effects of Generative AI. *arXiv Preprint arXiv:2403.01964*.

Kremer, M. (1993). The O-Ring Theory of Economic Development. *The Quarterly Journal of Economics 108*(3), 551–575.

McElheran, K., M.-J. Yang, E. Brynjolfsson, and Z. Kroff (2025). The Rise of Industrial AI in America. *SSRN Working Paper, No. 5036270*.

Meyer, A. N., T. Fritz, G. C. Murphy, and T. Zimmermann (2014). Software Developers' Perceptions of Productivity. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*.

Nagle, F. (2019). Open Source Software and Firm Productivity. *Management Science 65*(3), 1191–1215.

Noy, S. and W. Zhang (2023). Experimental Evidence on the Productivity Effects of Generative Artificial Intelligence. *Science 381*(6654), 187–192.

Otis, N., R. Clarke, S. Delecourt, D. Holtz, and R. Koning (2026). The Uneven Impact of Generative AI on Entrepreneurial Performance. *Management Science*.

Peng, S., E. Kalliamvakou, P. Cihon, and M. Demirer (2023). The Impact of AI on Developer Productivity: Evidence from GitHub Copilot. *arXiv Preprint arXiv:2302.06590*.

Petersen, K. (2011). Measuring and Predicting Software Productivity: A Systematic Map and Review. *Information and Software Technology 53*(4), 317–343.

Reimers, I. and J. Waldfogel (2026). AI and the Quantity and Quality of Creative Products: Have LLMs Boosted Creation of Valuable Books? *NBER Working Paper, No. 34777*.

Restrepo, P. (2025). We Won't be Missed: Work and Growth in the AGI World. *NBER Working Paper, No. 34423*.

Sarkar, S. K. (2026). AI Agents and Higher-Order Work. *SSRN Working Paper, No. 5713646*.

Sarkar, S. K. and L. Melas-Kyriazi (2026). Returns to Intelligence. *SSRN Working Paper, No.*

6578939.

Shephard, R. W. (1953). *Cost and Production Functions*. Princeton University Press.

Solow, R. M. (1987). We'd Better Watch Out. *New York Times Book Review*, 36.

Song, F., A. Agarwal, and W. Wen (2025). The Impact of Generative AI on Collaborative Open-Source Software Development: Evidence from GitHub Copilot. *Information Systems Research*.

Tomlinson, K., S. Jaffe, W. Wang, S. Counts, and S. Suri (2025). Working with AI: Measuring the Applicability of Generative AI to Occupations. *arXiv Preprint arXiv:2507.07935*.

Wright, N. L., F. Nagle, and S. Greenstein (2023). Open Source Software and Global Entrepreneurship. *Research Policy 52*(9), 104846.

Yotzov, I., J. M. Barrero, N. Bloom, P. Bunn, S. J. Davis, K. M. Foster, A. Jalca, B. H. Meyer, P. Mizen, M. A. Navarrete, P. Smietanka, G. Thwaites, and B. Z. Wang (2026). Firm Data on AI. *NBER Working Paper, No. 34836*.

Zhou, E. and D. Lee (2024). Generative Artificial Intelligence, Human Creativity, and Art. *PNAS Nexus 3*(3), pgae052.

Writing Code vs. Shipping Code: Productivity Effects Across Generations of AI Coding Tools

Mert Demirer Leon Musolff
Liyuan Yang

Appendix - For Online Publication

Contents

A Proofs and Derivations . . . . . **OA - 3**
A.1 Closed Form of the Nested CES Production Function OA - 3
A.2 Derivation of the Pass-Through Formula OA - 3
A.3 Proof of Proposition 1 OA - 4
A.4 Proof of Corollary 1 OA - 4
A.5 Proof of Proposition 2 OA - 5
A.6 Connection to Jones (2026) OA - 6

B Alternative Models of Attenuation . . . . . **OA - 7**
B.1 Decreasing Returns without Complementarity OA - 7
B.2 Decreasing Returns with Complementarity OA - 9
B.3 Modeling Reallocation of Inputs across Layers OA - 10
B.4 Simulated Attenuation Patterns OA - 13
B.5 Taking the Alternative Models to the Data OA - 14

C Data Appendix . . . . . **OA - 17**
C.1 Overview OA - 17
C.2 Coding Tool Usage Data OA - 17
C.3 Construction of the Control Pool OA - 24
C.4 Construction of Outcome Variables OA - 25
C.5 Data Collection Procedure OA - 30
C.6 GitHub Aggregate Data OA - 33
C.7 Software Marketplace Data OA - 34

D Estimation Details . . . . . **OA - 38**
D.1 Overview OA - 38
D.2 Matching Strategy OA - 38
D.3 Construction of the Estimation Sample OA - 40
D.4 Event Study Specification OA - 41
D.5 Heterogeneity by Pre-Period Activity OA - 43
D.6 Other Event-Study Specifications OA - 44
D.7 Aggregate Series OA - 46
D.8 Production Function Calibration OA - 46

E Robustness Checks . . . . . **OA - 48**
E.1 Robustness to Matching OA - 48
E.2 Robustness to Empirical Specification OA - 49
E.3 Ruling Out Activity Bias with Placebos OA - 51
E.4 External Validity: Public vs. Private Repositories OA - 52
E.5 Robustness to Activity Deletion OA - 52

F Time-Shifted Event Studies: Identifying Assumptions . . . . . **OA - 54**

G Additional Figures and Tables . . . . . **OA - 57**

A Proofs and Derivations

A.1 Closed Form of the Nested CES Production Function

Substituting recursively into the layer technology $y_s = [\alpha_s\, y_{s-1}^{\rho_s} + (1-\alpha_s)\, e_s^{\rho_s}]^{1/\rho_s}$ for $s \geq 2$, starting from $y_1 = e_1$, the final output $Y \equiv y_S$ admits the nested CES closed form

$$Y = \Bigg[\alpha_S \Bigg[\alpha_{S-1} \Big[\cdots \Big[\alpha_3 \big[\alpha_2 e_1^{\rho_2} + (1-\alpha_2) e_2^{\rho_2}\big]^{\rho_3/\rho_2} + (1-\alpha_3) e_3^{\rho_3}\Big]^{\rho_4/\rho_3} \cdots \Big]^{\rho_{S-1}/\rho_{S-2}} + (1-\alpha_{S-1}) e_{S-1}^{\rho_{S-1}}\Bigg]^{\rho_S/\rho_{S-1}} + (1-\alpha_S) e_S^{\rho_S}\Bigg]^{1/\rho_S},$$

where $\rho_s = (\sigma_s - 1)/\sigma_s$ and each effective input e_s is either $B_s h_s^p$ (augmentation) or $\min(a_s, \phi_s h_s^r)$ (partial automation), as defined in Section 3. The technology has $S-1$ levels of CES nesting (one per layer $s = 2, \ldots, S$), with elasticity of substitution σ_s governing the nest at layer s.

A.2 Derivation of the Pass-Through Formula

Consider a single layer of production combining upstream input x with effective effort e via a CES technology:

$$y = [\alpha\, x^{\rho} + (1-\alpha)\, e^{\rho}]^{1/\rho}, \qquad \rho = \frac{\sigma - 1}{\sigma}, \tag{12}$$

where $\sigma > 0$ is the elasticity of substitution and $\alpha \in (0,1)$ is the share parameter. In the context of the main model, $x = y_{s-1}$ and $e = e_s$. The output elasticity of upstream input is

$$\theta \equiv \frac{\partial \ln y}{\partial \ln x} = \frac{\alpha\, x^{\rho}}{\alpha\, x^{\rho} + (1-\alpha)\, e^{\rho}}. \tag{13}$$

A productivity shock multiplies upstream input by $A > 1$, so $\tilde{x} = Ax$, while effective effort e is unchanged. The new output is

$$\tilde{y} = [\alpha\, A^{\rho}\, x^{\rho} + (1-\alpha)\, e^{\rho}]^{1/\rho}.$$

Dividing by y and using $\alpha\, x^{\rho} = \theta\, [\alpha\, x^{\rho} + (1-\alpha)\, e^{\rho}]$:

$$\frac{\tilde{y}}{y} = \left[\frac{\alpha\, x^{\rho} A^{\rho} + (1-\alpha)\, e^{\rho}}{\alpha\, x^{\rho} + (1-\alpha)\, e^{\rho}}\right]^{1/\rho} = [\theta\, A^{\rho} + (1-\theta)]^{1/\rho}, \tag{14}$$

which is the formula in Equation (8). The pass-through depends only on the output elasticity θ and the elasticity of substitution σ; the share parameter α enters only through θ.

A.3 Proof of Proposition 1

Let $f(A, \theta, \sigma) \equiv [\theta\, A^{\rho} + (1-\theta)]^{1/\rho}$ with $\rho = (\sigma-1)/\sigma$ denote the single-layer pass-through.

Proof of Proposition 1. *(a)* $\partial f/\partial\sigma > 0$. The pass-through $f(A, \theta, \sigma) = [\theta A^{\rho} + (1-\theta)]^{1/\rho}$ is a power mean of A and 1 with weights θ and $1-\theta$ and order $\rho = (\sigma-1)/\sigma$. Power means are strictly increasing in their order, and ρ is strictly increasing in σ, so $\partial f/\partial\sigma > 0$ for all $A > 1$ and $\theta \in (0, 1)$.

(b) $\partial f/\partial\theta > 0$.

$$\frac{\partial f}{\partial \theta} = \frac{1}{\rho}\,[\theta\, A^{\rho} + (1-\theta)]^{\frac{1}{\rho}-1}\,(A^{\rho} - 1)\,.$$

When $\rho > 0$ ($\sigma > 1$), $A^{\rho} > 1$ and $1/\rho > 0$, so $\partial f/\partial\theta > 0$. When $\rho < 0$ ($\sigma < 1$), $A^{\rho} < 1$ and $1/\rho < 0$, so the product is again positive. In the Cobb–Douglas limit ($\rho \to 0$), $f = A^{\theta}$ and $\partial f/\partial\theta = A^{\theta} \ln A > 0$. □

A.4 Proof of Corollary 1

Proof of Corollary 1. Let $f_j(x) = [\theta_j\, x^{\rho_j} + (1-\theta_j)]^{1/\rho_j}$ denote the single-layer pass-through at layer j, with $\theta_j \in (0, 1)$ and $\rho_j = (\sigma_j - 1)/\sigma_j$; in the Cobb–Douglas case ($\rho_j \to 0$), $f_j(x) = x^{\theta_j}$.

(i) For any $x > 1$, $f_j(x)$ is a weighted power mean of x and 1 with strictly positive weights θ_j and $1-\theta_j$, so it lies strictly between its arguments: $1 < f_j(x) < x$. (Strictness of attenuation is where the requirement that effective input have positive weight at every layer, $\theta_j < 1$, is used; $\theta_j > 0$ ensures the gain remains strictly positive. The Cobb–Douglas case satisfies the same inequalities since $1 < x^{\theta_j} < x$.) Holding inputs at all other layers fixed, a shock $A > 1$ at layer k raises output at layer $s > k$ by the composition of the intervening pass-throughs,

$$\frac{\tilde{y}_s}{y_s} = f_s \circ f_{s-1} \circ \cdots \circ f_{k+1}(A),$$

which follows by applying Equation (14) recursively, treating the gain inherited from layer $j-1$ as the upstream shock to layer j. Since each application of f_j maps its argument $x > 1$ into $(1, x)$, the cumulative gain is strictly decreasing in s and, in particular, strictly smaller than the effect A at layer k for every $s > k$.

(ii) Each f_j is strictly increasing on $(0, \infty)$: $f_j'(x) = \theta_j\, x^{\rho_j - 1}[\theta_j\, x^{\rho_j} + (1-\theta_j)]^{1/\rho_j - 1} > 0$. The final-output gain from a technology entering at layer k is $E_k = f_S \circ \cdots \circ f_{k+1}(A)$, and

from entering at layer $k' > k$ it is $E_{k'} = f_S \circ \cdots \circ f_{k'+1}(A)$. By part (i), $f_{k'} \circ \cdots \circ f_{k+1}(A) < A$. Applying the strictly increasing composite map $f_S \circ \cdots \circ f_{k'+1}$ to both sides of this inequality yields $E_k < E_{k'}$. □

A.5 Proof of Proposition 2

Proof of Proposition 2(a). The proposition considers the symmetric case with common σ and θ at every layer. Let $L_s = \ln G_s$, so $R_s = L_{s+1}/L_s$. We show that R_s is monotone in s with the sign claimed in the proposition.

Step 1: Pointwise elasticity. The elasticity of the single-layer pass-through $f(x) = [\theta x^\rho + (1-\theta)]^{1/\rho}$ is

$$\eta(x) = \frac{\partial \ln f}{\partial \ln x} = \frac{\theta\, x^\rho}{\theta\, x^\rho + (1-\theta)},$$

with $\partial\eta/\partial x = \theta(1-\theta)\rho\, x^{\rho-1}/[\theta x^\rho + (1-\theta)]^2$. The sign of this derivative is the sign of ρ: η is strictly decreasing in x when $\sigma < 1$ and strictly increasing when $\sigma > 1$.

Step 2: Average elasticity. Define $\bar\eta(x) = \ln f(x)/\ln x$ for $x > 1$. Since $f(1) = 1$, $\ln f(x) = \int_0^{\ln x} \eta(e^u)\, du$, so $\bar\eta(x)$ is the running mean of $u \mapsto \eta(e^u)$ over $[0, \ln x]$. Differentiating:

$$\bar\eta'(x) = \frac{\eta(x) - \bar\eta(x)}{x \ln x}.$$

When $\sigma > 1$, $\eta(e^u)$ is strictly increasing in u, so the running mean lies strictly below the endpoint and $\bar\eta'(x) > 0$. When $\sigma < 1$, the inequality reverses and $\bar\eta'(x) < 0$.

Step 3: Composition. By construction, $R_s = \ln f(G_s)/\ln G_s = \bar\eta(G_s)$. Since G_s is strictly decreasing in s (Corollary 1):

- $\sigma < 1$: $\bar\eta$ decreasing, G_s decreasing $\Rightarrow$ R_s strictly increasing in s.
- $\sigma = 1$: $\bar\eta \equiv \theta \Rightarrow R_s = \theta$ for all s.
- $\sigma > 1$: $\bar\eta$ increasing, G_s decreasing $\Rightarrow$ R_s strictly decreasing in s.

□

Proof of Proposition 2(b). Taking $A \to \infty$ in the pass-through formula (14):

- **Complements** ($\sigma < 1$, $\rho < 0$): as $A \to \infty$, $A^\rho \to 0$, so

$$\left.\frac{\tilde y}{y}\right|_{A\to\infty} = (1-\theta)^{\frac{\sigma}{\sigma-1}} < \infty. \tag{15}$$

- **Cobb–Douglas** ($\sigma = 1$, $\rho \to 0$): $\tilde y/y = A^\theta \to \infty$.
- **Substitutes** ($\sigma > 1$, $\rho > 0$): $A^\rho \to \infty$, so $\tilde y/y \to \infty$.

□

A.6 Connection to Jones (2026)

The bound in Equation (15) coincides with the automation bound in Jones (2026), with the share parameter θ in the role of Jones's spending share. Substituting $\sigma = 1/2$ ($\rho = -1$) gives exponent $\sigma/(\sigma - 1) = -1$, so

$$(1 - \theta)^{\sigma/(\sigma-1)} = (1 - \theta)^{-1} = \frac{1}{1 - \theta},$$

which matches Jones's formula with θ in the role of his spending share. More generally, the bound at $\theta = 0.5$ varies sharply with σ:

σ	$\sigma/(\sigma - 1)$	Bound $(1 - \theta)^{\sigma/(\sigma-1)}$	Percent increase
0.25	$-1/3$	1.26	26%
0.50	-1	2.00	100%
0.75	-3	8.00	700%

With strong complements ($\sigma = 0.25$), even infinite upstream input raises output by at most 26%. At $\sigma = 0.5$, the illustrative case in Jones (2026), the maximum gain is a doubling. The bound increases sharply as σ approaches the Cobb–Douglas knife-edge ($\sigma = 1$), where it disappears entirely.

B Alternative Models of Attenuation

This appendix develops three alternative models to the model in the main text. Each retains the hierarchical structure of Section 3.1 and changes either the production technology or the developer's choice of inputs. For each we derive the implied pass-through and the resulting attenuation result.

B.1 Decreasing Returns without Complementarity

Production proceeds through S layers as in the baseline model, with the technology replaced by Cobb–Douglas:

$$y_1 = e_1, \qquad y_s = y_{s-1}^{\gamma_s} e_s^{\delta_s}, \qquad \gamma_s, \delta_s > 0, \qquad s = 2, \ldots, S, \tag{16}$$

with final output $Y = y_S$. Define the *scale elasticity* and the *relative upstream share*

$$\mu_s \equiv \gamma_s + \delta_s \leq 1, \qquad \bar{\theta}_s \equiv \frac{\gamma_s}{\gamma_s + \delta_s} \in (0, 1), \qquad \text{so that} \qquad \gamma_s = \bar{\theta}_s \, \mu_s. \tag{17}$$

Constant returns ($\mu_s = 1$) recover the baseline technology in its $\sigma_s \to 1$ limit, with $\gamma_s = \bar{\theta}_s$ playing the role of the baseline's share; $\mu_s < 1$ is the new parameter. Effective input e_s is produced from developer effort and AI as in Equation (3). The output elasticity of layer s with respect to upstream output is

$$\theta_s \equiv \frac{\partial \ln y_s}{\partial \ln y_{s-1}} = \gamma_s = \bar{\theta}_s \, \mu_s. \tag{18}$$

Under constant returns it is a pure share; under decreasing returns it is the product of the share and the scale elasticity. This decomposition is the source of every difference below.

Proposition 3 (Attenuation under decreasing returns)**.** *Consider a productivity gain that raises output at layer k by a factor $A > 1$, holding effective input at every layer above fixed. Then for $s > k$,*

$$\frac{\tilde{y}_s(A)}{y_s} = A^{\Gamma_{k,s}}, \qquad \Gamma_{k,s} = \prod_{j=k+1}^{s} \theta_j = \prod_{j=k+1}^{s} \bar{\theta}_j \, \mu_j, \tag{19}$$

and the gain at layer s is strictly smaller than the gain at layer k, since $\theta_j = \bar{\theta}_j \mu_j < 1$ at every intervening layer.

Proof. Taking logs in Equation (16) and holding e_s fixed gives $\ln \tilde{y}_s - \ln y_s = \gamma_s(\ln \tilde{y}_{s-1} - \ln y_{s-1})$ at each layer. Iterating from $\ln \tilde{y}_k - \ln y_k = \ln A$ yields Equation (19). Since $\bar{\theta}_j < 1$ and $\mu_j \leq 1$, we have $\theta_j < 1$ at every layer, and $\Gamma_{k,s} < 1$ follows. □

Remark 1 (Attenuation without a human bottleneck). The attenuation result of Corollary 1 of the main model survives, but with its hypothesis weakened. In the baseline model, a layer attenuates because additional upstream output must be combined with the downstream effective input; the weight of that input, together with its complementarity with upstream output, determines the strength of the attenuation. Under decreasing returns, a layer attenuates even when the effective input carries little weight and there is no complementarity—indeed even with no effective input at all ($\bar{\theta}_j = 1$)—with substantial attenuation driven by decreasing returns alone.

Remark 2 (Shape of the decline). With parameters common across layers, $\ln G_s = (\bar{\theta}\mu)^{s-k} \ln A$, so the survival rate defined in Proposition 2 is

$$R_s = \frac{\ln G_{s+1}}{\ln G_s} = \bar{\theta}\,\mu, \tag{20}$$

constant across layers and invariant to the size of the shock. Decreasing returns therefore change how much of the gain survives, but not the pattern in which it is lost: the proportional decline is the same at every layer, regardless of the size of the gain that reaches it or its position in the hierarchy.

Remark 3 (Transmission of infinite productivity effects). Proposition 2(b) shows that under complements infinite productivity effects are not transmitted to final output: even an unbounded improvement in AI delivers a finite gain, because human input at each layer remains essential. Under decreasing returns, by contrast, infinite productivity effects are transmitted: pass-through is $A^{\Gamma_{k,S}}$ with $\Gamma_{k,S} > 0$, so final output grows without bound as $A \to \infty$.

Remark 4 (Identification). With decreasing returns, pass-through across a layer is $A^{\bar{\theta}_s \mu_s}$, so attenuation is driven by the product of the two parameters: the observed attenuation identifies this product rather than the share and the scale elasticity separately.

Remark 5 (Allocation invariance). Under cost minimization with a Cobb–Douglas technology, the allocation of expenditure across inputs is determined by the technology parameters alone and does not depend on input prices. As a result, the allocation of human input does not change in response to AI productivity gains as long as input prices remain the same. The assumption in the main model (Section 3.4) that the input allocation is fixed—which we relax in Appendix B.3—is therefore without loss of generality under a Cobb–Douglas production function.

B.2 Decreasing Returns with Complementarity

This model incorporates decreasing returns into the CES production function, allowing us to consider complementarity together with decreasing returns; it nests both the baseline model and the model in Appendix B.1. The technology is the CES aggregator of Equation (2) with scale elasticity μ_s:

$$y_s = \left[\alpha_s\, y_{s-1}^{\rho_s} + (1-\alpha_s)\, e_s^{\rho_s}\right]^{\mu_s/\rho_s}, \qquad \rho_s = \frac{\sigma_s - 1}{\sigma_s}, \qquad 0 < \mu_s \leq 1, \tag{21}$$

which nests everything considered so far: $\mu_s = 1$ is the baseline model, $\sigma_s \to 1$ is Appendix B.1, and both together give the constant-returns Cobb–Douglas benchmark. Writing the share of the CES aggregator attributable to upstream output as

$$\bar{\theta}_s \equiv \frac{\alpha_s\, y_{s-1}^{\rho_s}}{\alpha_s\, y_{s-1}^{\rho_s} + (1-\alpha_s)\, e_s^{\rho_s}} \in (0,1), \tag{22}$$

the output elasticity again factorizes into share times scale, $\theta_s \equiv \partial \ln y_s / \partial \ln y_{s-1} = \mu_s \bar{\theta}_s$, as in the Cobb–Douglas production function considered in Appendix B.1.

Proposition 4 (Attenuation with complementarity and decreasing returns)**.** *A productivity gain raising y_{s-1} by a factor $A > 1$, with e_s held fixed, passes through as*

$$\frac{\tilde{y}_s(A)}{y_s} = \left[\bar{\theta}_s\, A^{\rho_s} + (1-\bar{\theta}_s)\right]^{\mu_s/\rho_s} = \left(G_s^{\mathrm{CRS}}(A)\right)^{\mu_s}, \tag{23}$$

where G_s^{CRS} denotes the baseline pass-through in Equation (8). Each layer attenuates strictly whenever $\bar{\theta}_s < 1$ or $\mu_s < 1$, and pass-through is strictly increasing in $\bar{\theta}_s$, in σ_s, and in μ_s.

Proof. Dividing post- and pre-gain output in Equation (21) gives Equation (23). Since $G_s^{\mathrm{CRS}}(A) < A$ whenever $\bar{\theta}_s < 1$ (a weighted power mean of A and 1), we have $(G_s^{\mathrm{CRS}})^{\mu_s} < A$ when either inequality is strict. Monotonicity in $\bar{\theta}_s$ and σ_s follows from Proposition 1 because $x \mapsto x^{\mu_s}$ is increasing; monotonicity in μ_s follows from $G_s^{\mathrm{CRS}} > 1$. □

Remark 6 (Comparison with the baseline model). Equation (23) makes the comparison with the baseline model transparent: the pass-through under complementarity and decreasing returns is the baseline pass-through raised to the power μ_s. The two forces compound, and each attenuates the gain for a different reason: the share channel ($\bar{\theta}_s < 1$) operates as in the baseline model, since additional upstream output must be combined with the downstream effective input, while the scale channel ($\mu_s < 1$) takes a further proportional cut of whatever survives, because additional throughput runs into decreasing returns. Across layers, the

two play distinct roles. Taking logs in Equation (23) gives $\ln G_{s+1} = \mu \ln G^{\text{CRS}}(G_s)$, so the survival rate is $R_s = \mu\, R^{\text{CRS}}(G_s)$, where R^{CRS} is the baseline survival function. The shape classification of Proposition 2(a) is therefore preserved exactly—R_s is increasing in s when $\sigma < 1$, constant at $\mu\bar{\theta}$ when $\sigma = 1$, and decreasing when $\sigma > 1$—so complementarity governs the shape of the attenuation profile, while decreasing returns shift its level from $\bar{\theta}$ to $\mu\bar{\theta}$ without changing the sign of its slope.

Remark 7 (Identification). With decreasing returns, what can be recovered from pass-through data changes. Expanding Equation (23) in $x = \ln A$,

$$\ln G(A) = \underbrace{\mu\,\bar{\theta}}_{\text{slope}}\; x \;+\; \frac{1}{2}\underbrace{\mu\,\rho\,\bar{\theta}(1-\bar{\theta})}_{\text{curvature}}\; x^2 \;+\; O(x^3). \tag{24}$$

The slope identifies only the product $\mu\bar{\theta}$, and the curvature-to-slope ratio only the combination $\rho\,(1-\bar{\theta})$: to this order, $(\bar{\theta}, \sigma, \mu)$ are not separately identified; separating them rests on higher-order curvature of the pass-through. Since the baseline model imposes $\mu_s = 1$, its estimated output elasticities recover the product $\mu_s\bar{\theta}_s$, which is a lower bound on the true upstream share, with equality if and only if returns to scale are constant.

B.3 Modeling Reallocation of Inputs across Layers

In this section, we relax the assumption of the baseline model that the allocation of developer effort across layers is fixed. Because reallocation is irrelevant under a Cobb–Douglas technology (Remark 5), we work with the CES technology of Equation (2). The model is the same as in the main text, with the only difference that the developer chooses her inputs at every layer, subject to a fixed total expenditure, as AI changes productivity. Characterizing this choice requires working with input prices, which we have abstracted from so far.

Since augmentation and automation are perfect substitutes in producing effective input, $e_s = B_s h_s^p + \min(a_s,\ \phi_s h_s^r)$, the cost-minimizing developer uses whichever is cheaper. Let w denote the value of the developer's time and p_s the price of a unit of AI output at layer s, the direct (inference) cost of AI output in Section 3.1. Under augmentation, one unit of effective input requires $1/B_s$ units of developer effort and so costs w/B_s. Under automation, the Leontief structure implies that at a cost minimum no AI output goes unreviewed ($a_s = \phi_s h_s^r$), so one unit of effective input requires $1/\phi_s$ units of review time together with one unit of AI output, and costs $w/\phi_s + p_s$. Effective input at layer s is therefore available at the constant unit cost

$$c_s = \min\left\{\frac{w}{B_s},\ \ \frac{w}{\phi_s} + p_s\right\}. \tag{25}$$

The developer's total expenditure on production, the value of her time plus her spending on AI output, is fixed at $\bar{E}$ and is allocated across the S layers:

$$w \sum_{s=1}^{S} \left(h_s^p + h_s^r \right) + \sum_{s=1}^{S} p_s \, a_s = \bar{E}. \tag{26}$$

The developer chooses her inputs to maximize final output subject to Equation (26); under constant returns, this is equivalent to producing any given output at minimum cost (developer time plus AI expenditure). Because the technology is homothetic, behavior is summarized by the unit cost function, given by $C_1 = c_1$ and the CES dual of Equation (2),

$$C_s = \left[\alpha_s^{\sigma_s} \, C_{s-1}^{1-\sigma_s} + (1-\alpha_s)^{\sigma_s} \, c_s^{1-\sigma_s} \right]^{\frac{1}{1-\sigma_s}}, \qquad s = 2, \ldots, S. \tag{27}$$

Define the upstream cost share at baseline prices,

$$\theta_s \equiv \frac{\alpha_s^{\sigma_s} \, C_{s-1}^{1-\sigma_s}}{\alpha_s^{\sigma_s} \, C_{s-1}^{1-\sigma_s} + (1-\alpha_s)^{\sigma_s} \, c_s^{1-\sigma_s}} \in (0,1), \tag{28}$$

which coincides with the output elasticity in Equation (9) evaluated at the optimal allocation, since cost shares equal output elasticities under constant returns. Under constant returns the minimum cost of producing Y units of final output is $C_S Y$, so the budget in Equation (26) pins down final output as $Y = \bar{E}/C_S$.[34]

Proposition 5 (Attenuation under reallocation)**.** *Consider an AI improvement that lowers the unit cost of effective input at layer* 1 *by a factor* $A > 1$*, and let the developer reoptimize the entire allocation in Equation* (26)*, holding total expenditure* $\bar{E}$ *fixed. The unit cost of producing layer-*s *output falls by the factor* G_s*, given by* $G_1 = A$ *and*

$$G_s = \left[\theta_s \, G_{s-1}^{\sigma_s - 1} + (1-\theta_s) \right]^{\frac{1}{\sigma_s - 1}}, \qquad s = 2, \ldots, S, \tag{29}$$

and final output rises by the same factor G_S*.*

Proof. Only C_{s-1} changes at step s, so from Equation (27), $(C_s')^{1-\sigma_s} = \alpha_s^{\sigma_s}(C_{s-1}/G_{s-1})^{1-\sigma_s} + (1-\alpha_s)^{\sigma_s} c_s^{1-\sigma_s} = C_s^{1-\sigma_s} \left[\theta_s G_{s-1}^{\sigma_s - 1} + (1-\theta_s) \right]$. Taking the ratio to $C_s^{1-\sigma_s}$, raising to $1/(1-\sigma_s)$, and inverting gives Equation (29); induction on s completes the argument. Strict

[34] If instead total developer time were fixed at H and AI output were purchased at the prices p_s without a budget constraint, final output would rise by the factor by which developer time per unit of final output ($\partial C_S / \partial w$ by Shephard's lemma) falls, which equals G_S in Proposition 5 if and only if the developer-time share of unit cost is unchanged by the improvement, for instance when AI output carries no monetary cost.

attenuation follows because G_s is a weighted power mean of G_{s-1} and 1, hence lies strictly between them whenever $\theta_s < 1$. Finally, $Y = \bar{E}/C_S$ before and $Y' = \bar{E}/C'_S$ after the improvement, so $Y'/Y = C_S/C'_S = G_S$. □

The G_s are unit-cost (productivity) indices, and because total expenditure is fixed, G_S is also the realized gain in final output. Measured *intermediate* quantities additionally reflect the reallocation of expenditure across layers: the simulations in Appendix B.4 compute these equilibrium quantities via Shephard's lemma.

Remark 8 (Comparison with the baseline model). Similar to the baseline model (Corollary 1), attenuation is strict at every layer with $\theta_s < 1$, so the attenuation results are robust to the reallocation of inputs. However, different from the baseline model, reallocation replaces every layer's curvature $\rho_s = (\sigma_s - 1)/\sigma_s$ with $\sigma_s - 1$. This replacement makes attenuation weaker: for every layer and every $A > 1$, $G_s > G_s^{\text{fixed}}$ (unless $\sigma_j = 1$ at every layer $j \le s$), since the map $x \mapsto [\theta_s G^x + (1-\theta_s)]^{1/x}$ is a weighted power mean and hence nondecreasing in x, and $(\sigma_s - 1) - \rho_s = (\sigma_s - 1)^2/\sigma_s \ge 0$. This is the Le Chatelier principle (Samuelson, 1947; Milgrom and Roberts, 1996): allowing an input to adjust cannot lower the response of output.

Remark 9 (Equivalent elasticity). Because the two recursions differ only in the curvature, pass-through under frictionless reallocation has the fixed-allocation functional form with σ_s replaced by the equivalent elasticity

$$\tilde{\sigma}_s = \frac{1}{2 - \sigma_s} \qquad (\sigma_s < 2). \tag{30}$$

Reallocation therefore makes adjacent layers behave as though more substitutable ($\sigma_s < \tilde{\sigma}_s < 1$ for $\sigma_s \in (0,1)$), but never converts complements into substitutes, since $\tilde{\sigma}_s \gtrless 1$ if and only if $\sigma_s \gtrless 1$. Note also that $\tilde{\sigma}_s \ge 1/2$ for every $\sigma_s \ge 0$, with $\tilde{\sigma}_s \downarrow 1/2$ as $\sigma_s \downarrow 0$: even a Leontief technology behaves, once the developer reoptimizes, as one with an elasticity of one half. This again makes clear that our results on the role of complementarity in attenuation are robust to reallocation.

Remark 10 (Direction of reallocation). If inputs are reallocated, from which layers and to which layers do they move? Applying Shephard's lemma to Equation (27), expenditure on a layer's own effective input relative to its upstream input responds to an AI improvement with an elasticity governed by σ_s. Under complements ($\sigma_s < 1$) better AI shifts expenditure, and with it developer effort, downstream, toward the human-bottlenecked layer; under substitutes it shifts them toward the layer where AI operates.

B.4 Simulated Attenuation Patterns

The three models above, and their combinations, imply different attenuation patterns across the hierarchy. Using the results derived in the previous subsections, we now simulate the attenuation pattern of a productivity gain at the first layer, under the four technologies listed below:

Specification	Substitution	Scale
Cobb–Douglas, constant returns	$\sigma = 1$	$\mu = 1$
Cobb–Douglas, decreasing returns	$\sigma = 1$	$\mu = 0.9$
CES complements, constant returns	$\sigma = 0.5$	$\mu = 1$
CES complements, decreasing returns	$\sigma = 0.5$	$\mu = 0.9$

Common parameters are a baseline upstream share of $\bar{\theta} = 0.7$ at every layer, and a gain of $A = 2$ (a doubling of effective coding input).[35] Each specification is solved twice, giving eight combinations in total. Under *fixed allocation*, the baseline ($A = 1$) optimal allocation is held fixed when the gain arrives, as in Section 3.4. Under *reallocation*, the developer reoptimizes the entire allocation of inputs after the gain, holding total expenditure fixed, as in Appendix B.3.[36]

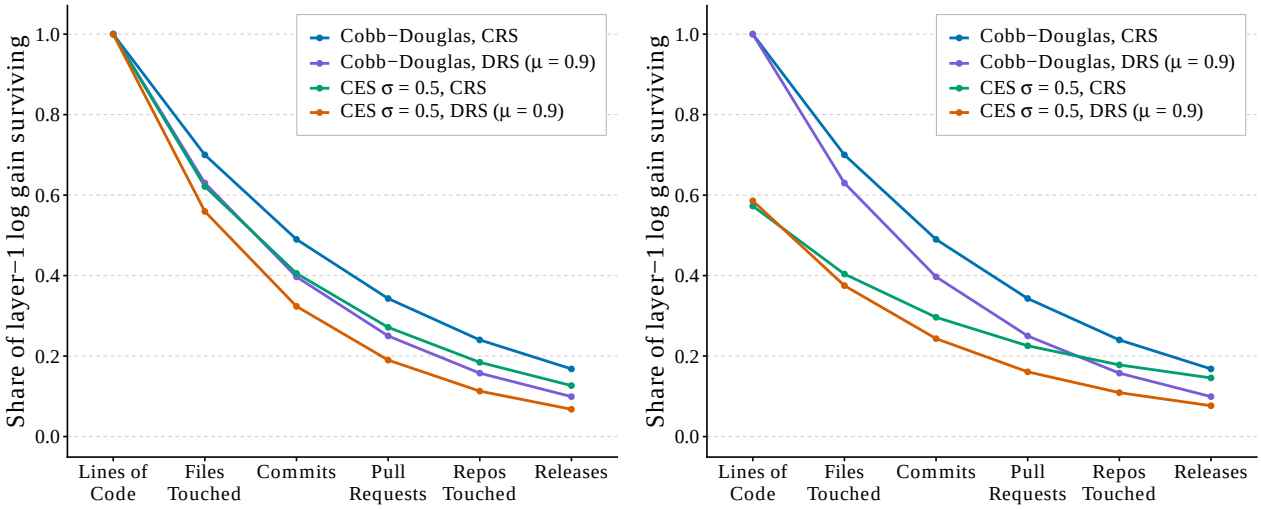
Figure OA-1 plots, at every layer, the share of the layer-1 log gain surviving in measured layer output, $\ln(\tilde{y}_s/y_s)/\ln A$, with fixed allocation in Panel (a) and reallocation in Panel (b). Starting with Panel (a), the main pattern is that every specification attenuates, and steeply: only a small fraction of the layer-1 log gain reaches releases. Decreasing returns deepen the decline roughly uniformly, lowering survival at every layer under both technologies while leaving the shape of the profile unchanged, consistent with Propositions 3 and 4. The specifications are therefore distinguished less by how much attenuation they produce than by its shape across layers: the Cobb–Douglas per-layer survival rate is constant, whereas under CES complements it rises going up the hierarchy. Complementarity front-loads the attenuation, while decreasing returns spread it evenly.

Panel (b) isolates the role of reallocation. Under Cobb–Douglas the profiles are identical across panels, as Remark 5 implies: optimal effort shares do not depend on the size of the gain, so allowing reallocation changes nothing. Under CES, reallocation compresses the profile from both ends. The profiles in Panel (b) begin below one, because expenditure

[35] For the CES specifications the weights α_s are calibrated so that the baseline share equals 0.7 layer by layer; for Cobb–Douglas the exponents are $\gamma = 0.7\mu$ and $\delta = 0.3\mu$.

[36] The simulations measure effective input at each layer in units whose unit cost is one at baseline and treat the gain as a fall in the unit cost of layer-1 input, so they hold for any split of that cost between developer time and AI output.

Figure OA-1: Simulated Attenuation Across the Production Hierarchy



(a) Fixed allocation **(b)** With reallocation

Notes: The figure plots the share of the layer-1 log gain surviving at each layer of the hierarchy, $\ln(\tilde{y}_s/y_s)/\ln A$, following a doubling of effective coding input ($A = 2$) at layer 1. Each line is one of the four technologies described in the text. Panel OA-1(a) holds the developer's allocation of inputs across layers fixed at its pre-gain optimum; Panel OA-1(b) lets the developer reoptimize the allocation after the gain. In Panel OA-1(b) the two Cobb–Douglas lines coincide with their Panel OA-1(a) counterparts.

reallocates from the coding layer to other layers, so the measured gain at the lines-of-code layer understates the actual productivity gain. At the top of the hierarchy the ranking reverses: the reallocation profiles end above their fixed-allocation counterparts. Reallocation gives up measured gains low in the hierarchy in order to deliver more final output, which is the Le Chatelier result of Appendix B.3.

Overall, these simulation results show that our main results on attenuation, and its relationship with complementarity, are robust to these alternative specifications. The main effect of the alternative technologies is on the shape of the attenuation profile. In the next section, we use our empirical attenuation estimates to assess how well these alternative models fit the empirical pattern.

B.5 Taking the Alternative Models to the Data

Appendix B.4 shows that these models imply different attenuation patterns. We now use our empirical estimates to assess how well each of them fits the data. To measure the fit of the empirical pattern to each model, we repeat the estimation of Section 6.3, minimizing the sum of squared log deviations between predicted and empirical gains, for all eight specifications considered in the previous section—{Cobb–Douglas, CES} × {constant returns, decreasing returns} × {fixed allocation, reallocation}. As in the main text, each specification is calibrated on autocomplete, because autocomplete delivers a

Table OA-1: Parameter Estimates and Fit across Production Function Specifications

Technology	Allocation	$\bar{\theta}$	σ	μ	$\hat{A}_{\text{Auto}}$	Loss ($\times 10^{-4}$)
Cobb–Douglas, CRS	fixed	0.428	–	1	3.34	295.2
Cobb–Douglas, CRS	realloc.	0.428	–	1	3.34	295.2
Cobb–Douglas, DRS	fixed	$\bar{\theta}\mu = 0.428$		–	3.34	295.2
Cobb–Douglas, DRS	realloc.	$\bar{\theta}\mu = 0.428$		–	3.34	295.2
CES, CRS	fixed	0.762	0.229	1	3.34	2.5
CES, CRS	realloc.	0.591	0.306	1	41.05	9.7
CES, DRS	fixed	0.981	0.128	0.702	3.34	1.2
CES, DRS	realloc.	0.769	0.123	0.778	1,349	1.3

Notes: Loss is the sum of squared log deviations between predicted and empirical autocomplete gains across all six layers ($\times 10^{-4}$). $\hat{A}_{\text{Auto}}$ is the layer-1 gain: observed under fixed allocation (the lines gain, $1 + 234.3/100$) and estimated under reallocation. The four Cobb–Douglas rows are a single model: allocation invariance makes the reallocation cells identical, and only $\bar{\theta}\mu$ is identified, so the decreasing-returns cells re-estimate the same profile. The reallocation specifications hold total expenditure fixed, as in Appendix B.3. The CES specification with constant returns and reallocation is computed in closed form using the recursive CES dual and Shephard's lemma.

productivity gain at the first layer only, which allows us to use its attenuation across the layers above.

Two specifications carry a caveat. First, under Cobb–Douglas the share and the scale elasticity are not separately identified, only their product $\mu\bar{\theta}$ (Remark 4). Second, under reallocation the layer-1 gain (A) becomes a further free parameter, because the measured lines response is an equilibrium outcome—the gain net of the expenditure the developer moves out of coding—rather than the gain itself. The reallocation specifications therefore carry an additional estimated parameter, $\hat{A}_{\text{Auto}}$.

Table OA-1 reports the results. The four Cobb–Douglas rows report identical parameters and losses: reallocation is irrelevant under Cobb–Douglas (Remark 5), and attenuation data identify only the product $\bar{\theta}\mu$ (Remark 4), so all four cells re-estimate the same profile. The Cobb–Douglas specifications are inconsistent with the curvature of the profile. Their best fit has a loss more than 100 times that of the baseline CES specification (295.2 against 2.5, in units of 10^{-4}), and the failure is the one Appendix B.1 anticipates: the attenuation pattern in Figure 8 is too steep to be explained by a constant rate of decline.

Adding decreasing returns to the CES specification improves the loss, from 2.5 to 1.2. However, this improvement comes with a set of parameters that are unstable and implausible: the share runs to its boundary at $\bar{\theta} = 0.981$, and the returns to scale become too small ($\mu = 0.702$). We therefore conclude that the CES specification with decreasing returns to scale is not a good fit for the data. Even if it were the true model, however, it would imply stronger complementarity than our baseline estimate ($\sigma = 0.128$ against 0.229), reinforcing rather than weakening our main conclusion.

The two reallocation specifications fit worse than their fixed-allocation counterparts, despite the additional free parameter that reallocation introduces. The comparison is like-for-like because every specification is fit to the same six layers. Under fixed allocation the layer-1 term is degenerate, since A is set to the observed lines gain; under reallocation it is not, because the measured lines gain is itself an equilibrium object that the model must fit, and it is this term that pins $\hat{A}$. More importantly, they fit only by assuming a much larger initial productivity shock than is observed: the estimated layer-1 gain is $\hat{A} = 41.05$ under constant returns and $\hat{A} = 1{,}349$ under decreasing returns, against a measured lines gain of 3.34. Gains of this magnitude are not plausible for autocomplete. This suggests that the model with reallocation is not a good fit to the estimated pattern, likely because of the limited scope for reallocation in the short run.

Taken together, the results in this appendix suggest that the main mechanism highlighted by our model survives in the extensions considered here. Moreover, the observed attenuation pattern provides empirical support for the baseline specification used in the paper.

C Data Appendix

This appendix describes the data underlying our empirical analysis. We first give an overview of the sample construction and then explain each step in detail.

C.1 Overview

Our sample consists of GitHub developers and their coding activity. It includes two groups of users: developers who adopted one of the AI coding tools we study (*treated* users) and an untreated control pool from which matched controls are drawn. The remainder of this appendix proceeds in two steps. First, we describe how we construct the set of treated and control developers. Second, we explain how we construct the outcome variables measuring these developers' coding activity, aggregated into developer-week panels, including how we identify the activity of async coding agents.

C.2 Coding Tool Usage Data

We study seven different coding tools: five AI coding tools and two non-AI comparison tools (Docker and GitHub Pro), used as placebo and bounding exercises. The identification of the adoption of these tools comes from separate sources. For the GitHub Agent, OpenAI Codex, Claude Code, and Docker, we identify adoption from traces of activity on GitHub—dedicated bot usernames, branch-name prefixes, or specific configuration files. For GitHub Autocomplete, GitHub Sync, and GitHub Pro, we use internal data from Microsoft and identify adoption from users' first activity or subscription to the product. We now provide the data collection details for each tool.

C.2.1 GitHub Agent

The GitHub Agent operates by creating pull requests that are authored by the bot account `copilot-swe-agent[bot]`. The commits within these pull requests are co-authored by the human developer who assigned the issue or requested the change.[37]

To identify users, we use the GitHub REST API issues endpoints to retrieve all pull requests created by the GitHub Agent, identified by the author `copilot-swe-agent[bot]`. We then use the GitHub REST API endpoint for listing commits on a pull request to retrieve all commits in each PR. For each pull request opened by the agent, we assign developer users according to the following priority hierarchy:

1. **Co-authors**: We extract co-author information from commit messages by parsing the "Co-authored-by:" trailer, which follows the pattern `Co-authored-by: Name`

[37] GitHub Docs—About Coding Agent

Table OA-2: Identification of Adopters of GitHub Agent

	PR-level		User-level	
User source	Count	Percent (%)	Count	Percent (%)
coauthor	451,936	65.08	64,350	58.64
requested reviewer	189,946	27.35	38,842	35.39
assignee	50,950	7.34	6,554	5.97
Missing	1,575	0.23	—	—
Total	694,407	100.00	109,746	100.00

Notes: This table reports the distribution of how human users are linked to GitHub Agent pull requests. We apply a priority hierarchy that searches for co-authors first, then requested reviewers, and finally assignees (Section C.2.1). The PR-level columns count the 694,407 GitHub Agent pull requests in our data; the User-level columns count the 109,746 distinct human users identified across these PRs. "Missing" denotes PRs for which no human user could be assigned under any of the three sources; this category is empty at the user level by construction.

`<ID+login@users.noreply.github.com>`. When multiple co-authors are identified within a single PR, we include all unique co-author IDs.

2. **Requested reviewers**: For PRs without co-author information, we use the requested reviewer IDs from the pull request metadata.

3. **Assignees**: For PRs with neither co-authors nor requested reviewers, we use the assignee IDs from the pull request metadata.

We exclude the Copilot bot account itself from the user list. All but 1,575 pull requests (0.23% of the total) are linked to one or more human users according to the hierarchy above; pull requests with no identifiable human user are excluded. We define the event date as the creation date of the earliest associated Copilot pull request for each user.

As of 2025-12-22, we retrieved 694,407 pull requests opened by the GitHub Agent and 2,084,472 commits associated with these pull requests. Applying the priority hierarchy above, we identify 109,746 distinct users. Table OA-2 reports the user-source distribution, showing that the majority of users are identified through the co-author field (65.08% of pull requests and 58.64% of users) and that most of the remainder are identified through requested reviewers, with assignees accounting for only a small share. Figure 3(c) shows the distribution of event dates.

C.2.2 OpenAI Codex

We use the GitHub REST API endpoints for issues to retrieve all pull requests created by Codex, identified by the head branch prefix `head:codex/`. We define a user as the creator of a Codex pull request, as recorded in the API response's `user` field. As of 2025-12-22,

we retrieved 2,643,226 pull requests created by 142,902 Codex users. We exclude the very few pull requests (21 in total) with creation dates outside the feasible observation window—either before 2025-05-16 (the release date of Codex[38]) or after the data collection date. Figure 3(b) shows the distribution of event dates.

C.2.3 Claude Code

Unlike the asynchronous agents, sync agents leave no distinct trace of their coding activity in our data: their output is committed by the developer under their own account and is indistinguishable from code written without assistance. Claude Code, however, stores project-specific memory and instructions in a `CLAUDE.md` file, which it automatically pulls into its context when starting a conversation.[39] Because this file is prominent in the workflow and tends to be created soon after a developer begins using the tool, we take the appearance of a `CLAUDE.md` file as an indicator of Claude Code adoption. Accordingly, we use the GitHub REST API code search endpoint (`/search/code`) to search for files named `CLAUDE.md` within repositories owned by users in our user universe.[40]

For each discovered `CLAUDE.md` file, we use the GitHub REST API commits endpoint (`/repos/{owner}/{repo}/commits?path={path}`) to retrieve the complete commit history for that file. We define a user as a Claude user if they appear as an author of any `CLAUDE.md` commit, and we assign each such author's earliest `CLAUDE.md` commit date as the event date.

We identify 59,614 `CLAUDE.md` files across 51,865 repositories, along with 179,879 associated commits authored by 28,558 unique authors. We exclude users whose event dates fall before 2025-02-24 (the release of Claude Code) or after the data collection date. Figure 3(a) shows the distribution of event dates.

C.2.4 GitHub Sync

We observe AI tool usage data for GitHub Copilot subscribers from Microsoft, covering April 2025 through December 2025. These data record usage at the level of the individual interaction—a prompt for the sync agent and a completion for autocomplete—together with the tool used, which allows us to identify the first time each developer uses the sync

[38] On May 16, 2025, OpenAI announced the launch of a research preview of Codex, based on a finetuned version of OpenAI o3. Wikipedia—OpenAI Codex

[39] Anthropic—Claude Code Best Practices

[40] On 2024-12-06, we queried GitHub user accounts by ID via the GitHub REST API, up to the highest ID assigned as of that date (190,801,309), and retrieved user information for 174,130,663 IDs (91.26%). Missing observations largely reflect deleted, suspended, or otherwise unavailable accounts. Next, using GHArchive (GHArchive), we require each ID's first appearance (as the actor in any event type) to occur before 2020-01 and the last appearance to occur after 2022-12. This restriction yields 5,499,509 users (2.88% of the total), which constitutes the user universe.

Table OA-3: Analysis Windows by Tool

Tool	Data source	Treated window	Last adoption date
GitHub Autocomplete	Microsoft	01/22 – 12/22	2022-12-26
GitHub Sync	Microsoft	05/25 – 12/25	2025-11-30
Claude Code	Public GitHub	02/25 – 12/25	2025-12-22
GitHub Agent	Public GitHub	05/25 – 12/25	2025-12-22
OpenAI Codex	Public GitHub	05/25 – 12/25	2025-12-22
GitHub Pro	Microsoft	01/22 – 12/22	2022-12-26
Docker	Public GitHub	04/22 – 12/25	2025-12-19

Notes: "Data source" is where the adoption signal comes from: confidential Microsoft records or public GitHub activity. "Treated window" is the calendar window over which adoptions enter the analysis; matched controls are drawn from the same window shifted one year earlier. "Last adoption date" is the latest adoption date among matched treated developers.

functionality of GitHub Copilot. Because a developer already using the tool when our data begin would be misclassified as a new adopter, we discard the first six weeks of the panel and restrict attention to developers whose first observed usage falls after mid-May 2025, taking each developer's first observed usage as the adoption date. To isolate sync agent usage from async agent usage, we further restrict this sample to users who never use the GitHub Agent. This yields a universe of 74,296 users with adoption dates spanning 2025-05-15 to 2025-11-30.

These data cover all GitHub Copilot users, not only those who are active on GitHub. Many of these developers therefore leave no trace in our GitHub activity data and cannot be used in our analysis; we drop them at a later stage of the sample construction. Table OA-4 reports the resulting counts.

C.2.5 GitHub Autocomplete

We obtain from Microsoft a universe of 2,985,745 individual subscribers to GitHub Copilot; 397,586 of these have subscription dates in 2022, spanning 2022-06-21 to 2022-12-26. At the time, Copilot offered only autocomplete functionality, so we treat these subscribers as autocomplete adopters. Unlike the sync agent, this dataset does not record actual usage, so we take the subscription date as the adoption date.

C.2.6 GitHub Pro

GitHub Pro is a paid individual subscription that extends the free GitHub plan. It provides advanced collaboration and code-review features on private repositories—required and multiple pull request reviewers, protected branches, and code owners—together with repository insight graphs, wikis and GitHub Pages on private repositories, higher GitHub

Table OA-4: Treated-Sample Construction by Tool

Tool	Unconditional	In window	With commit data	History floor	Active at adoption	After top-1% activity trim	Matched
				Panel A. Users			
GitHub Autocomplete	2,985,745	397,586	336,451	261,701	15,483	15,328	15,328
GitHub Sync	74,296	74,296	69,668	57,898	7,613	7,537	7,537
Claude Code	28,390	28,390	27,464	26,373	—	26,110	25,612
GitHub Agent	109,746	109,746	103,211	85,310	—	84,458	84,458
OpenAI Codex	142,899	142,899	132,313	90,047	—	89,146	89,146
GitHub Pro	22,632	22,260	21,911	16,628	1,340	1,326	1,050
Docker	494,529	315,390	308,194	278,765	—	275,977	266,265
Total	3,858,237	1,090,567	999,212	816,722	504,931	499,882	489,396
				Panel B. Commits			
GitHub Autocomplete	—	—	25,021,718	24,212,012	5,498,472	4,957,828	4,957,828
GitHub Sync	—	—	4,908,695	4,168,459	1,145,464	1,035,579	1,035,579
Claude Code	—	—	7,281,516	7,169,802	—	6,122,851	5,954,146
GitHub Agent	—	—	14,274,347	13,056,775	—	10,987,245	10,987,245
OpenAI Codex	—	—	14,326,320	10,665,321	—	9,058,390	9,058,390
GitHub Pro	—	—	2,025,870	1,778,730	448,739	406,326	328,027
Docker	—	—	105,305,360	102,406,074	—	91,209,248	87,328,376
Total	—	—	173,143,826	163,457,173	140,390,647	123,777,467	119,649,591
				Panel C. Pull Requests			
GitHub Autocomplete	—	—	2,203,485	2,135,496	1,013,794	972,142	972,142
GitHub Sync	—	—	730,955	621,838	279,589	266,869	266,869
Claude Code	—	—	897,557	881,639	—	795,026	776,312
GitHub Agent	—	—	1,933,137	1,792,215	—	1,570,358	1,570,358
OpenAI Codex	—	—	3,315,743	2,241,929	—	2,080,853	2,080,853
GitHub Pro	—	—	190,381	174,481	81,223	78,398	61,829
Docker	—	—	12,879,320	12,382,211	—	11,306,082	10,725,149
Total	—	—	22,150,578	20,229,809	18,672,600	17,069,728	16,453,512

Notes: Panel A reports the number of adopters remaining at each construction stage; Panels B and C report the commits and pull requests made within the tool's treated window (Table OA-3) by the adopters remaining at each stage. The "Unconditional" column counts all users identified from the tool's adoption signal, and "In window" those whose adoption date falls within the tool's treated window. "With commit data" keeps the users present in the collected commit data, dropping accounts whose login no longer resolves (deleted or suspended), accounts at bot scale (more than 100,000 commits in the collection window or more than 1,000 on a single day), for which complete retrieval is infeasible under API limits, and adopters with no collected commits; commit and pull request volumes exist only from this stage onward, because activity histories were collected for in-window adopters only (Section C.5). "History floor" requires at least one week with positive commits at or before event week −11, so that the pre-adoption window used for matching contains measurable history, and "Active at adoption" requires both a commit and a pull request in the adoption week; the latter applies only to the subscription-based tools, whose Microsoft adoption dates need not coincide with public GitHub activity ("—" where not applicable; column totals treat it as non-binding). "After top-1% activity trim" drops the top 1% of eligible adopters by pre-period commit volume, and "Matched" counts the adopters for whom a match could be found in the control pool (see Appendix D for donor-pool eligibility and the matching procedure); matched controls equal matched treated by construction. Users adopting several tools appear once per tool.

Actions and Codespaces allowances, and email support.[41] It provides no AI functionality; Appendix E.3 discusses its use as a placebo.

Similar to GitHub Copilot, we obtain a universe of individual subscribers to GitHub Pro from Microsoft, comprising 22,632 users with subscription dates spanning 2022-01-01 to 2022-12-31. As with autocomplete, we do not observe usage in these data and take the subscription date as the adoption date.

C.2.7 Docker

Docker is a widely used open-source platform for containerization: it packages an application together with its dependencies and system configuration into a *container* that runs identically on any machine. Developers configure a container through a `Dockerfile`, a plain-text file committed to the repository that specifies the steps required to build the container image. The appearance of a `Dockerfile` in a repository therefore marks the point at which a developer begins using Docker for that project, in the same way that a `CLAUDE.md` file marks the adoption of Claude Code; Appendix E.3 discusses its use as a placebo and bounding exercise.

We therefore study developers who commit `Dockerfile` files. We use the GitHub REST API code search endpoint (`/search/code`) with the query `filename:dockerfile` to search for files named "dockerfile" (case-insensitive) within repositories owned by users in our user universe. For each discovered `Dockerfile`, we retrieve the complete commit history and define a user as a `Dockerfile` user if they appear as a commit author. The event date is defined as the earliest `Dockerfile` commit date for each user.

As of 2025-12-22, we identify 1,942,751 `Dockerfile` files across 1,073,370 repositories, with 5,257,726 associated commits authored by 540,709 unique users. The earliest and latest event dates in these data are 2018-01-01 and 2025-12-19, respectively.

C.2.8 Sample Selection

Table OA-3 lists, for each tool, the data source, the treated window over which adoptions enter the analysis, and the last adoption date. For the AI tools released in 2025, the treated window ends with our data collection in December 2025 and begins with the tool's release, or, for GitHub Sync, with the start of reliable usage data (Table OA-3). We deliberately stop including adoptions after December 2025: control outcomes are measured one year earlier, so including 2026 adoptions would require drawing controls from 2025, and since the adoption of agentic AI tools became widespread from mid-2025 onward, we could no longer ensure that matched controls do not themselves use AI tools. For autocomplete,

[41] GitHub Docs—GitHub's Plans

we include only adoptions in 2022, after the product launched, to rule out contamination from other AI tools, such as the use of ChatGPT for coding. For GitHub Pro, we likewise use 2022 to rule out the use of AI tools. Docker adoption is included from 2022 through 2025 to give us enough statistical power.

Table OA-4 then summarizes how the sample changes as we apply our sample selection criteria. Panel A reports the number of users, and Panels B and C report the same for commits and pull requests. (For Codex, these counts exclude the 21 pull requests with creation dates outside the feasible observation window.) Each column applies the selection criteria progressively. "Unconditional" corresponds to all users identified in the raw adoption data. "In window" restricts the sample to adoptions within each tool's treated window, as described in Table OA-3. "With commit data" counts the users for whom we download commit data; this sample differs from the previous column for a few reasons: the account has since been deleted, the user has more than 100,000 commits in the collection window, or the user has more than 1,000 commits on any single day. We eliminate these accounts because such implausibly large volumes typically correspond to bot activity, and their complete retrieval is infeasible under GitHub REST API pagination and rate limits. "History floor" conditions on developers having activity at or before week −11 relative to adoption; we impose this restriction because we do not want to analyze the commit activity of users who joined the platform to try these tools, but rather condition on already-active developers.

"Active at adoption" drops users who are not active at the time of adoption. This restriction (both a commit and a pull request in the adoption week) is imposed only on the subscription-based tools, whose Microsoft adoption dates need not coincide with public GitHub activity; for Claude Code, the async agents, and Docker, adoption is itself public activity, and we do not impose the additional requirement (estimates are similar when we do; Appendix Table OA-11). As expected, this significantly reduces the sample sizes for GitHub Autocomplete, GitHub Sync, and GitHub Pro, since adoption of these tools is identified off-platform and an adopter may have little or no public GitHub activity. "After top-1% activity trim" drops the top 1% of adopters by pre-period commit volume, removing extremely active users whose volumes, again, typically correspond to bots. Finally, we match the remaining users to control developers, and "Matched" reports the number of matched users. The matching procedure is described in Appendix D, and the order in which these data were collected in Section C.5.

Panels B and C report the corresponding volumes of commits and pull requests. These panels start from the "With commit data" column, since activity volumes can be computed only for the users whose commit data we download. Across the selection steps, the decline

Table OA-5: Control-Sample Construction

Stage	Users	Commits	Pull requests
GHArchive actor accounts, 2021–2025	67,934,433	—	—
Selected for collection	23,446,654	—	—
Login resolved	22,642,849	—	—
With commit data	20,795,563	2,662,776,922	—
With commit data and ≥1 pull request	8,211,358	1,856,691,155	195,398,318
Matched control (any tool)	434,982	251,384,683	53,535,493

Notes: This table traces the construction of the control pool. The "Selected for collection" row counts all 13,446,654 accounts with at least twenty pushes plus the stratified sample of 10,000,000 accounts with fewer (Section C.5), and "Login resolved" drops deleted or suspended accounts. "With commit data" requires at least one collected commit, excluding accounts at bot scale (more than 100,000 commits in the collection window or more than 1,000 on a single day) and accounts with no commits in 2020–2026; requiring in addition at least one pull request makes a developer eligible for a donor pool. "Matched control" counts the distinct developers matched to at least one treated adopter under the production specification; a control developer can serve several tools, so the matched counts of Table OA-4 sum to 489,396, above the 434,982 distinct developers here. Commits and pull requests are totals over 2020-01-01 through 2026-05-31 for the users at each stage.

in activity is much smaller than the decline in users, which is expected since the restrictions primarily eliminate users who are not active on the platform. For autocomplete, for example, the number of users falls from 336,451 to 15,328—a decline of roughly 95%—but commits fall only from 25,021,718 to 4,957,828 (roughly 80%), and pull requests from 2,203,485 to 972,142 (roughly 56%). Our sample selection therefore eliminates a large fraction of users while retaining a disproportionately large share of the coding activity.

C.3 Construction of the Control Pool

To match each treated user to a control, we draw a set of candidate users from a broad universe of GitHub users. We next describe this procedure.

We first enumerate all GitHub accounts. In August 2026, we walked the GitHub REST API users endpoint (`https://api.github.com/users?since=1&per_page=100`) over the entire account-id space, retrieving every existing account: 275,822,614 accounts in total, of which 256,043,973 are user accounts and 11,253,874 are organizations, with the remainder bot and mannequin accounts. We use this enumeration to characterize the universe of GitHub accounts; the control pool itself is constructed from GHArchive, as described below.

Ideally, this entire universe would serve as the control pool. However, because we match on commit activity, we would need each user's commit history, and downloading commit data for all 256 million user accounts is infeasible under GitHub API rate limits. We instead use GHArchive (GHArchive), a complete public log of GitHub events that can be downloaded in bulk and records every user's push events. Measuring activity directly

from it is unreliable, however, especially toward the end of our sample period (Appendix Figure OA-18), so we use GHArchive only as a screening tool to select a set of users for whom we download commit data and which constitutes the control pool.

We select users as follows. Since treated users are AI adopters, who tend to be active developers, we first include all accounts with twenty or more push events over 2021–2025 (13,446,654 accounts); this captures a relatively active set of developers. To also represent less active developers, we add a stratified random sample of the 54,487,779 accounts with fewer than twenty pushes: within each exact push count $n = 0, \ldots, 19$, we draw a fixed number of accounts, declining linearly from 900,000 accounts at $n = 0$ to 100,000 at $n = 19$ (10,000,000 accounts in total, drawn without replacement with a fixed seed). Because we set the sampling rates ourselves (3.0% of zero-push accounts up to 27.6% of nineteen-push accounts), population aggregates can be recovered with inverse-probability weights. Finally, we resolve each selected account's numeric `id` to its current `login` through the REST API; 13,152,488 of the ≥20-push accounts and 9,490,361 of the sampled low-activity accounts resolve, and the remainder are deleted or suspended. The combined control pool contains 22,642,849 accounts.

Table OA-5 traces this construction, reporting at each stage the number of users and, where available, their corresponding commit and pull request volumes. We start from the 67,934,433 accounts that appear as event actors in GHArchive over 2021–2025. Applying the selection procedure above, we select 23,446,654 accounts for collection, of which 22,642,849 resolve to a current login; the remainder are deleted or suspended accounts. Of these, 20,795,563 have commit data: the gap reflects accounts with no commits over 2020–2026 as well as accounts at bot scale, which we exclude. Requiring at least one pull request—the condition for entering a donor pool—leaves 8,211,358 users, of whom 434,982 distinct developers are ultimately matched to at least one treated adopter.

C.4 Construction of Outcome Variables

This section describes how we construct the outcome variables used in our analysis for both treated and control developers. The variables fall into two groups: those used in the matching process, which we call the commit list and the pull request list, and the outcomes used in the event study. We refer to the commit and pull request data used in matching as *lists* because they come from a different download process covering a different user pool; the corresponding event-study outcomes are simply called commits and pull requests. We go through each variable in turn and explain its data source.

C.4.1 *User-level Commit List*

We use the GitHub REST API List commits endpoint and filter requests by the author query parameter using each user's GitHub login to retrieve commit data between January 2020 and May 2026, separately for the treated and control groups. This endpoint returns commits on each repository's default branch, so the developer-level series covers main-branch activity only. Commits that an asynchronous agent makes on its own feature branch are collected separately, from the agents' pull requests (Section C.4.15), and are therefore captured even though they do not appear in this list.

In the commit lists we retrieve, the same commit can appear multiple times: when the repository containing the original commit is forked, a single commit—identical in every field, including its `sha`, except for the repository path—appears in the original repository and in each of its forks, and would be counted once per copy absent deduplication. We therefore deduplicate commits at the level of the commit `sha`, assigning each distinct `sha` a single canonical repository and discarding every other copy. While this procedure need not retain the repository in which the commit was originally made, the commit's date is preserved in the forked copies, so the choice does not affect the commit, line, or pull request outcomes; it introduces some small error in the outcomes keyed to repository identity—repositories touched, files touched (identified by repository–filename pairs), and releases, which are counted on the repositories a user's commits touch. Overall, 5.8% of unique commits appear in more than one copy and are collapsed to a single record in this way.

This procedure gives us the list of commits each user makes, together with their dates, from which we construct the time series of commits. Note that this procedure does not capture commits that users have since deleted; our evidence suggests, however, that deletion is not significant (see Appendix E.5).

C.4.2 *User-level Pull Request List*

For the treated users identified in Section C.2 and the control pool constructed in Section C.3, we also download the list of pull requests each user created. We use the GitHub Search API to query all pull requests authored by each user's login within the collection window. For each pull request we retrieve its `pull_request_url` (of the form `.../repos/{owner}/{repo}/pulls/{number}`), the creating user's `user_id`, and timestamps including `created_at`, `merged_at`, and `state`. We deduplicate pull requests by their `pull_request_url`.

C.4.3 Commits

We download the commit content only for the treated users whose adoption date falls within the analysis window, as defined in Section C.2.8, and, among controls, only for the matched control users. We impose this restriction because the commit endpoint returns one commit per call, making this the most expensive step of our data collection (Section C.5).

We download the content of every commit in each user's commit list. For each commit, we follow its `url` (of the form `.../repos/{owner}/{repo}/commits/{sha}`) to the GitHub REST API commit endpoint and retrieve the commit's metadata: its line statistics (total lines changed, additions, and deletions), the list of files it touches, and its commit message.

For users of the GitHub Agent and Codex async agents, we distinguish between human-authored and agent-authored commits, classifying commits as agent-authored when they occur within agent-initiated pull requests. Although human developers can, in principle, contribute commits to such pull requests, we observe that such cases are extremely rare. The two sets are disjoint by construction (a commit is agent-authored if and only if it lies in an agent-initiated pull request), and we use this decomposition to examine human coding activity and AI-generated code separately (Section C.4.15).

From each developer's commit list—which records, per commit, the commit `url`, the `commit_author_date`, and the `author_id`—we count the distinct commits (by `sha`, after deduplication) whose author date falls in each week.

C.4.4 Lines of Code

We measure the total number of lines of code each developer writes in each week, using the commit contents downloaded in Section C.4.3. Each commit's record reports its line statistics: the number of lines added (`stats_additions`), the number of lines deleted (`stats_deletions`), and their sum (`stats_total`), where a modified line counts as one addition and one deletion. To construct the weekly outcome, we sum `stats_total` across a developer's commits whose author date falls in each week.

C.4.5 Files Touched

We measure the number of distinct files each developer touches in each week, using the same commit contents. Each commit's record lists the files it changes with, for each file, its `filename`, `status`, and per-file `additions`, `deletions`, and `changes`. We identify a file by the pair (repository, `filename`) and, for the weekly outcome, count the distinct files appearing in a developer's commits within each week.

C.4.6 Pull Requests

We construct a panel of pull requests at the developer level, distinguishing between developer-initiated pull requests and agent-initiated pull requests for asynchronous agents: we assign each pull request to a week by its `created_at` and count the pull requests a developer creates each week, classifying each as agent-initiated or developer-initiated according to whether its `pull_request_url` appears in the Codex or Copilot agent pull-request lists. As discussed above, we use agent-initiated pull requests to identify adoption of async agents, and developer-initiated pull requests to measure developer activity over time.

C.4.7 Repositories Touched

We measure the number of distinct repositories touched by each developer in each week. This allows us to get a sense of the extent to which developers are working on multiple projects at once.

We parse the repository `{owner}/{repo}` from each commit `url`, assign each commit to its canonical repository after deduplication, and count the distinct repositories a developer commits to within each week.

C.4.8 Releases

Repositories on GitHub can be 'released' by their owners, which typically involves tagging a commit as a release, uploading related binaries, and posting a release note and/or changelog. Releases let us gauge the extent to which the repositories a developer works on are shipping versioned output to their users.

We collect releases at the repository level. For each repository in our sample we query the GitHub REST API releases endpoint (`/repos/{owner}/{repo}/releases`), retrieving results 100 at a time and paging through the returned releases. For each release we obtain the repository it belongs to, the date and time it was created, the user who authored it, its tag and title, and whether it is marked as a draft or a pre-release; for our measure we use only the repository and the creation date.

To build a developer-level weekly outcome, we date each release by the week in which it was created. We say a developer *touches* a repository in a given week if the developer makes at least one commit to that repository during that week. For each developer and each week, we take the set of repositories the developer touched in any of the twelve weeks ending in that week and count how many distinct repositories in that set had at least one release created during that week.

C.4.9 Lines Added

We measure the total number of lines each developer adds in each week, summing `stats_additions` (Section C.4.3) across the developer's commits whose author date falls in each week.

C.4.10 Lines Deleted

We measure the total number of lines each developer deletes in each week, summing `stats_deletions` across the developer's commits within each week.

C.4.11 Lines per Commit

We measure the average size of a developer's commits in each week, dividing the weekly sum of `stats_total` by the number of commits in that week. The outcome is defined only for weeks with at least one commit.

C.4.12 Files per Commit

We measure the average number of distinct files per commit in each week: the weekly average, across a developer's commits, of the number of distinct files each commit touches. The outcome is defined only for weeks with at least one commit.

C.4.13 Branches Created

We measure the number of branches each developer creates in each week. This outcome comes from a separate collection: for each repository in our sample we query the GitHub REST API activity endpoint (`/repos/{owner}/{repo}/activity`), paging through the returned events by cursor; each event records its type, the branch it refers to, its timestamp, and the acting user. We count, for each developer and week, the branch-creation events in which the developer is the actor. The API retains no events before March 2023, so this outcome exists only from 2023-03-07 onward, which is why it cannot be estimated for autocomplete, whose adoptions all occur in 2022 (Appendix D.6).

C.4.14 Branches Deleted

We measure the number of branches each developer deletes in each week, counting the branch-deletion events in which the developer is the actor. These events come from the same GitHub REST API activity endpoint used for branch creations, and are subject to the same March 2023 retention floor.

C.4.15 Asynchronous Agent Outcomes

To estimate async effects separately, we need to tell the asynchronous agents' activity apart from the developer's own. We do so for the two async agents we observe, the GitHub Agent and Codex, constructing agent-specific versions of five outcomes (commits, lines changed, files touched, repositories touched, and pull requests created): each is measured on the activity inside the agent's pull requests, with the developer's own activity measured outside them. Below we explain how agent activity is identified and attributed.

GitHub Agent The agent opens pull requests under its own bot account, `copilot-swe-agent[bot]`. We retrieve every pull request opened by this account, together with the commits each one contains, and count a pull request and its commits as agent activity only from the agent's public-release date (2025-05-19) onward. Since the bot, not the developer, is the committer of record, its commits must be reassigned to the human who initiated the work. The `Co-authored-by` trailers in each bot commit message list the collaborating developers' GitHub numeric identifiers; after excluding the agent's own two bot accounts (numeric identifiers 198982749 and 175728472), we create one record per named developer. Commits whose message is exactly "Initial plan" or "Initial plan for issue" are the agent's automated planning commits, and we drop them. When a bot commit names no co-author at all, we look to the other commits in the same pull request: if they point to exactly one developer, the commit is assigned to that developer; otherwise it is dropped.

Codex For Codex, no reassignment is needed, since its commits are recorded under the developer's own account. We identify Codex pull requests by the `codex/` prefix of their head branch and count them as agent activity from Codex's public-release date (2025-05-16) onward.

With agent activity identified and attributed, each outcome is computed three times per developer-week: once on commits inside the developer's GitHub-agent pull requests, once on commits inside their Codex pull requests, and once on commits outside any agent pull request. Pull requests created are split along the same lines into agent-initiated and developer-initiated, so agent activity never enters the developer's own totals.

C.5 Data Collection Procedure

This subsection summarizes the order in which we collect the data described above and the reasoning behind that order. The binding constraint throughout is the GitHub REST API's rate limit: *listing* endpoints return up to one hundred commits or pull requests per call, whereas *metadata* endpoints require one call per commit (for line counts and files changed) and several calls per repository (for branches, releases, and activity). Collecting per-

commit metadata for the full control universe would require billions of calls; collecting it only for matched controls requires hundreds of millions. The pipeline is therefore organized breadth-first: we first build cheap, complete user universes from bulk archives, then collect activity *lists* for those universes, then match, and only then collect the expensive per-commit and per-repository metadata, for the treated group in full and for the control group only where a developer was actually matched. Figure OA-2 visualizes the flow; the steps below give the details.

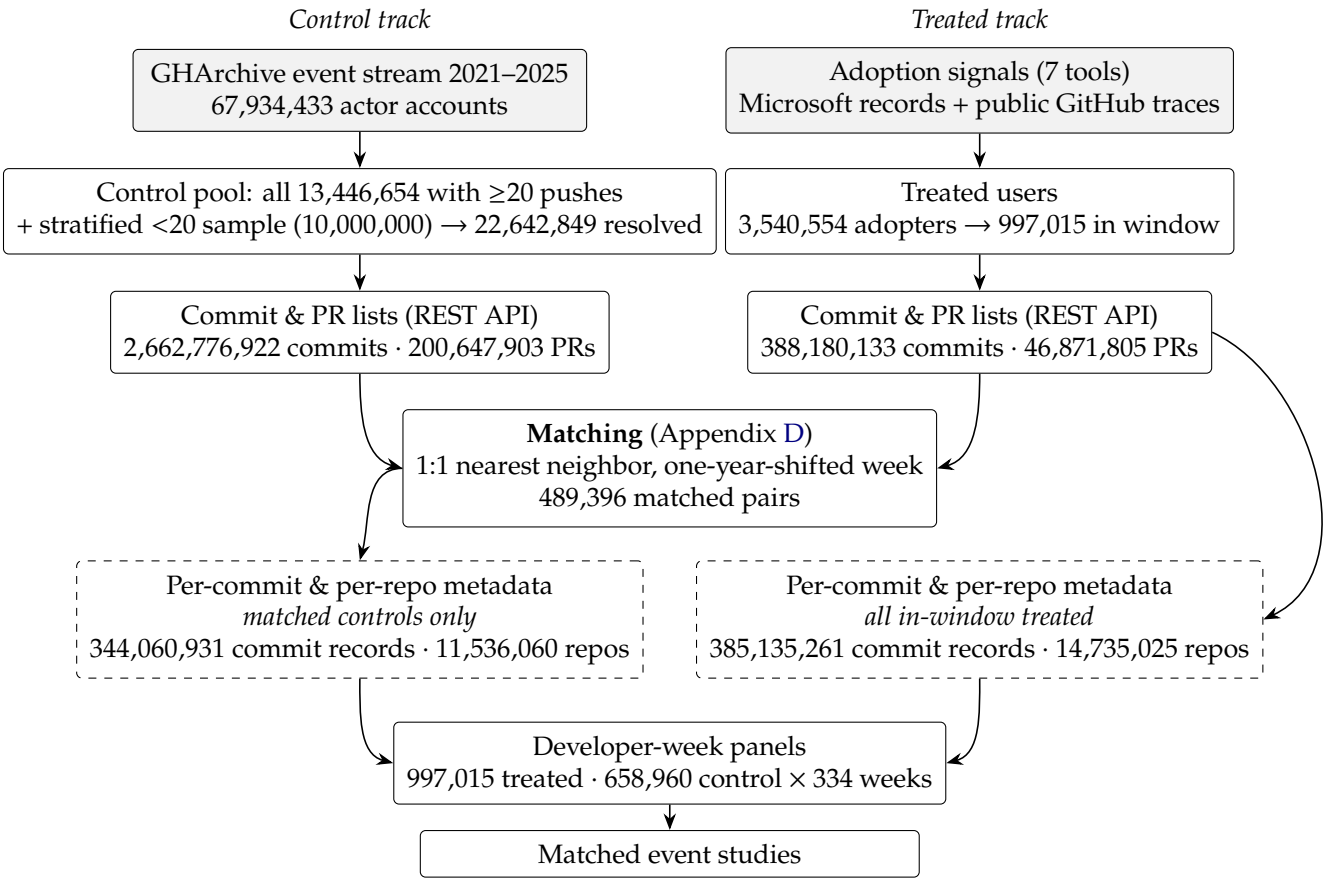
Step 1. Control pool from GHArchive. We download the complete GHArchive event stream for 2021–2025 and count push events for every account observed as an event actor (67,934,433 accounts in total). From these we build the control pool described in Appendix C.3: all 13,446,654 accounts with twenty or more pushes, plus a stratified sample of less active accounts, resolved to 22,642,849 current logins. We use GHArchive only for this screening step, not to measure activity, because its coverage of commit activity deteriorates after mid-2024.

Step 2. Treated users from adoption signals. The treated pool comes from the seven per-tool adoption signals described earlier in this appendix: Microsoft records for GitHub Autocomplete, GitHub Pro, and GitHub Sync; public GitHub traces for the GitHub Agent, Codex, Claude Code, and Docker. Together the signals identify 3,540,554 distinct adopters, of whom 997,015 adopt within their tool's analysis window (Table OA-3).

Step 3. Commit and pull request lists for both groups. For every user in both pools we retrieve the full commit list (commits-by-author endpoint) and pull request list over 2020-01-01 through 2026-05-31, excluding accounts at bot scale (more than 100,000 commits in total or 1,000 in a day), and deduplicate commits at the `sha` level because forks copy commit histories verbatim. The lists contain 2,662,776,922 unique control commits, 200,647,903 control pull requests by 8,474,057 users, 388,180,133 treated commits, and 46,871,805 treated pull requests. Treated users already covered by the control scrape are not re-fetched.

Step 4. Matching. Each in-window treated developer is then matched to a control developer (donor-pool eligibility and the matching procedure are described in Appendix D). This yields 489,396 matched pairs across the seven tools, built from 456,220 distinct treated developers and 434,982 distinct controls: matching is one-to-one and without replacement within a tool, but nothing prevents a control from being matched to adopters of several tools. Table OA-4 traces the treated sample tool by tool, from identified adopters down to the final estimation sample.

Figure OA-2: Data Collection Flow



Notes: The pipeline runs breadth-first. The control track (left) screens the full GitHub population through GHArchive push counts and collects commit and pull request *lists* for the resulting universe; the treated track (right) does the same for adopters identified from the seven tool signals. Matching operates on these lists. The expensive per-commit and per-repository metadata (dashed boxes) are then collected for all in-window treated developers but only for *matched* controls, which reduces the control-side metadata collection from roughly 2.7 billion API calls to 346,457,054.

Step 5. Per-commit metadata. Commit-level metadata (lines added and deleted, files changed, and the commit message) costs one API call per commit, which is why matching comes first. On the treated side we fetch all 388,180,133 commits (385,135,261 retrieved, a coverage of 99.2%); on the control side we fetch only the 346,457,054 commits belonging to matched controls (344,060,931 retrieved, 99.3%). Fetching metadata for the full control universe instead would have taken roughly 2.7 billion calls.

Step 6. Repository metadata. Every repository appearing in the collected commits (14,735,025 on the treated side, 11,536,060 on the control side) gets a further set of pulls: the repository record (creation date, default branch), the branch list, the release history, the activity feed (available from March 2023 onward), merge commits, and

the pull request behind each merge event. The release and branch outcomes in the developer-week panels are built from these inputs.

Step 7. Asynchronous agent footprint. Separately from the user-level collection, we download the async agents' own traces: every pull request opened by the GitHub Agent's bot account or on a `codex/` branch (6,551,030 pull requests), the commits inside each such pull request (14,554,797 distinct commits), their per-commit metadata, and, for the GitHub Agent, the pull request objects needed to attribute each agent PR to a human developer (1,840,294 attributed). These data let us separate agent-generated from human-generated output.

Step 8. Panels. Finally, we aggregate all collected activity into the developer-week panels described above: 997,015 treated and 658,960 control developers over 334 weeks.

Because matching precedes the control-side metadata collection, a change to the matching specification requires re-collecting metadata only for newly matched controls; the commit and PR lists of Step 3 cover the full universes and never need to be re-fetched.

C.6 GitHub Aggregate Data

This section describes the data behind the aggregate GitHub series of Figure 10. All three series are monthly and cover public GitHub repositories.

Claude Code Commit Share The numerator is the daily count of public commits authored or co-authored by Claude Code, identified from the commit signature the tool adds by default, as tracked by claudescode.dev; we aggregate the daily series to calendar months. The denominator is the total number of public commits per month, obtained from the GitHub Search API commit endpoint. The Search API reports estimated counts from a sampled index rather than exact totals, so the share is approximate. As discussed in the main text, it is in any case likely an undercount of AI-assisted activity, since developers often commit AI-generated work under their own name.

Pull Requests Merged Monthly counts of merged pull requests on public repositories, obtained from the GitHub Search API issues endpoint with one query per calendar month restricting to pull requests merged in that month.

New Repositories Created Monthly counts of newly created public repositories, obtained from the GitHub Search API repository endpoint with one query per calendar month on the repository creation date; repository search excludes forks by default, so the series counts non-fork repositories.

C.7 Software Marketplace Data

We collect data on four software marketplaces: the Google Play Store (Android), the Apple App Store (iOS), the Chrome Web Store, and SourceForge. For each marketplace we measure the number of apps available over time and an app-level measure of usage; the usage measure differs across stores. For each marketplace we obtain a near-complete catalog at regular intervals and, from the resulting sequence of snapshots, build a longitudinal app-level dataset that records each product's first and last observed dates, the months in which it appears, and the values of a set of marketplace-specific attributes. Because apps are periodically removed from the store, a single current snapshot's creation dates would understate entry; counting new apps therefore requires the full sequence of historical snapshots.

The raw data extend to our collection in August 2026 and are thus nearly real-time. In every panel, we restrict the usage analysis to entry cohorts from 2020-01 through 2026-05, so that each cohort's three-month usage window is fully observed; each app contributes one row per calendar month it is observed. Apps sometimes drop out of and reappear in the catalog between an app's first and last observed snapshot; for those interim months we back-fill attributes from the most recent prior snapshot. The remainder of this subsection describes each marketplace in turn.

C.7.1 Google Play Store (Android)

Our Google Play data come from SimilarWeb (previously known as 42matters), which crawls public store pages on a fixed schedule; we license its weekly snapshots, each a flat record per app per crawl.[42] The snapshots we obtain begin on 2020-02-29 and run through 2026-07-25 (335 in total). Coverage is worldwide in principle, but an app enters a given snapshot only if the crawl actually reached it that week.

The Play Store package name serves as our persistent app identifier; each record also names the publishing developer account and reports the app's category and whether it is an Application or a Game. Installs are the one awkward variable: Google discloses them only as ranges ("1,000–5,000", "5,000–10,000", and so on up to "10,000,000,000–50,000,000,000"), so we store both bounds and treat the lower one as a conservative measure of cumulative installs; most of our cohort-level summaries use it. Beyond that, a record carries a running version count, an indicator for an update since the prior snapshot, the average rating on the 1–5 scale, and the total number of ratings received.

An app's entry and exit dates are the earliest and latest snapshots in which its package

[42] See similarweb.com.

name appears; the resulting file is our universe of Play Store apps over the sample window. To build the monthly panel, we keep, for each app and calendar month, the record from the last weekly crawl of that month (so monthly variables always come from the same point within the month) and fill interim months in which an app temporarily leaves the store by carrying its most recent prior snapshot forward. The panel has 250,922,156 app–month rows for 9,815,182 unique Android apps.

Three limitations are worth keeping in mind: the install ranges make apps within a bucket indistinguishable; a weekly crawl can miss an app delisted and re-listed within the same week; and we observe only the worldwide catalog, not country-level availability.

C.7.2 Apple App Store (iOS)

We license weekly snapshots of the Apple App Store from the same vendor (SimilarWeb), spanning 2020-03-01 through 2026-07-26 across 334 weekly crawls. The data are collected in the same way as the Android snapshots, one record per app per snapshot.

Each record identifies the app by its App Store track identifier, our persistent id, and by the developer account that publishes it, and reports the primary genre and category, the content-advisory rating, the current price, an in-app-purchase indicator, the binary's size in bytes, and iPhone and iPad compatibility. The raw data list every country store in which the app is available; we retain the primary country, the number of countries, and an indicator for the U.S. store, together with counts of supported languages, supported devices, and distinct in-app features. Rating signals cover the lifetime average rating and total count, the same figures for the current version, and the one-to-five-star breakdown; we also record a per-snapshot update indicator and a running version count. Apple does not publicly disclose download or install counts, so the iOS panel contains no download variable.

The monthly panel is built exactly as for Android and contains 155,959,418 app–month rows covering 5,287,204 unique iOS apps. As on Android, weekly snapshots can miss short-lived listings; because Apple discloses no install counts, we measure adoption through ratings.

C.7.3 Chrome Web Store

We license daily snapshots of the Chrome Web Store from chrome-stats.com, a data provider that crawls the public Chrome Web Store on a fixed schedule.[43] We use two of its feeds. The *ranking-stats* feed is a daily snapshot of every item available in the store on the crawl date, with the item's full catalog metadata attached (publisher, category,

[43] See chrome-stats.com.

version, ratings, current user count, and the item-type label). The *all-versions* feed records each item's version-release history: every version ever published, with its release date. The all-versions feed carries no metadata beyond name, author, size, and description. Its release dates, however, extend back to the launch of the Chrome Web Store in 2009, far earlier than our crawl coverage of ranking-stats, which begins on 2023-03-31.

Combining the two feeds gives each item an entry and an exit date: entry is the earlier of the first release date in all-versions and the first appearance in ranking-stats, and exit is the last appearance in ranking-stats or, for items never observed there, the most recent release date in all-versions. All metadata fields (ratings, user count, category, publisher, etc.) come from ranking-stats and are therefore available only from 2023-03-31 onward; for the months before that, items have entry information from all-versions but no measured attributes.

Each ranking-stats record identifies the item by its Chrome Web Store id, our persistent identifier, and reports its type (extension, app, or theme), its category (e.g., Productivity, Developer Tools), the publishing developer account, the current user count, the average rating and rating count, a running version count, and an update indicator.

Apps have been deprecated by Google and no longer receive updated values, and themes are visual customizations rather than software, so we restrict the analysis to extensions.

Unlike Google Play, which reports a cumulative installation range, the Chrome Web Store reports a current user count at the time of each snapshot, which can move both up and down as users add or remove the extension. The reported count is exact for small extensions but rounded to bucket boundaries (10,000, 20,000, 30,000, . . . , 100,000, 200,000, . . . , 1,000,000, 2,000,000, and so on) for larger ones. To construct a measure comparable to the other marketplaces, we treat the running maximum of the per-snapshot user count over the observation window as cumulative installs.

The monthly panel is built as for Android and iOS, applied to the combined entry and exit dates above: one row per extension per observed month, pegged to the last ranking-stats snapshot of that month, with interim gaps back-filled from the most recent prior snapshot.

The main Chrome-specific limitation is the bucket rounding described above, which makes two large extensions in the same bucket indistinguishable.

C.7.4 SourceForge

We scrape the SourceForge project directory ourselves. The list of projects comes from SourceForge's published sitemap feeds at `https://sourceforge.net/sitemap-N.xml`,

which yield 328,614 distinct projects. For each project we then fetch the monthly download statistics from SourceForge's public JSON statistics endpoint over the window 2020-01 through 2026-08. We scrape the project's HTML overview page for its category and tag assignments, summary description, listed license, registration date, and publicly visible maintainer information. Finally, we pull the project's release RSS feed to obtain the dated history of file releases.

For each project we observe a stable slug, used as the persistent identifier, and the monthly download series (80 months). Metadata (category, tags, license, description, registration date) are fixed attributes, and the release feed gives one row per release with date, tag, and filename.

Download counts are reported at the project (not file) level and capture downloads from any SourceForge mirror, but exclude external mirrors and repackagings. The HTML metadata reflects the state of the project page at scrape time rather than a historical record, and the project enumeration is a single cross-section, so projects delisted from the sitemap before the scrape do not appear in the panel.

D Estimation Details

D.1 Overview

This appendix describes the estimation procedure underlying the matched difference-in-differences event studies in the main text. Section D.2 explains how we determine eligible control developers and form matched treated–control pairs; the remaining subsections define the estimation sample (Section D.3) and set out the difference-in-differences specifications, along with the inference and winsorization conventions used throughout, together with the constructions behind the remaining exhibits and the production-function calibration.

D.2 Matching Strategy

D.2.1 Design

For each tool, we pair every treated developer i with a control developer j observed one year earlier. We shift the adopter's adoption week, denoted t_i^*, back one calendar year; the shifted week, denoted t_i^p, is the control's *pseudo-adoption week*, and we select as the match a control developer j whose activity around t_i^p resembles the adopter's activity around adoption (Sections D.2.2 and D.2.3). In the estimation, the control's outcomes are aligned in event time so that the pseudo-adoption week plays the role of the adoption week.[44] Because the shift is exactly one year, seasonal patterns in developer activity (holidays, academic calendars, release cycles) stay balanced between the two groups. This matters because adoption dates for most tools span many months over which activity varies systematically. The per-tool treated and control windows are reported in Table OA-3.

To avoid activity bias, we would like both sides of a match to be actively coding at the (pseudo-)adoption week. To ensure this, we require every matched control to have at least one commit and at least one pull request in the pseudo-adoption week. Treated developers face the same requirement at adoption, except for the tools whose adoption we observe on public GitHub. For these the adoption event itself guarantees some public activity in the adoption week (a commit for Claude Code and Docker, a pull request for Codex, and, for the GitHub Agent, a pull request opened by the agent on the developer's behalf), and we impose no further requirement. Appendix Table OA-11 re-estimates the event studies imposing the full commit-and-pull-request requirement on all adopters, with similar results. The requirement therefore binds only for the tools identified from

[44] Weeks are the fixed seven-day blocks of the panel grid, anchored at 2020-01-01 (Appendix C). The adoption date is shifted back one calendar year (February 29 maps to March 1), and the pseudo-adoption week is the grid week containing the shifted date.

Microsoft records (GitHub Autocomplete, GitHub Pro, and GitHub Sync), whose adoption dates need not coincide with any public GitHub activity. Table OA-4 traces the resulting counts stage by stage.

D.2.2 Control Pool Eligibility

We start from the control pool constructed in Appendix C.3 (Table OA-5). A candidate control is eligible for a given adopter if it (i) never adopts the tool in question during its treated window (Table OA-3); (ii) has both a commit and a pull request in the pseudo-adoption week, the activity requirement of Section D.2.1; and (iii) has at least one week with positive commits at or before event week −11 relative to the pseudo-adoption week, mirroring the requirement imposed on treated adopters.

D.2.3 Matched-Pair Algorithm

We pair treated and control developers with a sequential greedy algorithm, processing treated developers in random order. For each treated developer i:

1. Retrieve the set of eligible controls $\mathcal{C}_i$ at the pseudo-adoption week t_i^p (Section D.2.2), removing any control already matched.
2. For each $j \in \mathcal{C}_i$, compute the absolute difference in pre-period commits $\Delta_{ij} = \left|\text{PreCommits}_i - \text{PreCommits}_j\right|$, where

$$\text{PreCommits}_i = \sum_{k=-11}^{-1} \text{commits}_{i,\, t_i^* + k}, \quad \text{PreCommits}_j = \sum_{k=-11}^{-1} \text{commits}_{j,\, t_i^p + k},$$

   and select the control with the smallest difference.
3. If several controls tie, one is chosen at random.[45]

Because many eligible controls share a treated developer's pre-period commit count, the realized difference Δ_{ij^*} is exactly zero for 99.8–100% of matched pairs. Matching is one-to-one and without replacement within each tool; a control developer may be matched in several tools' samples.

In practice, sequential matching never exhausts a donor pool: a treated developer goes unmatched only when no eligible control exists at t_i^p in the first place—10,486 of 499,882 adopters across the seven tools, mostly GitHub Pro (276 of 1,326 eligible adopters) and Docker (9,712 of 275,977).

[45] Both random orderings, the processing order over treated developers and the tie-breaking order, use fixed seeds, recorded for replication.

D.3 Construction of the Estimation Sample

Before estimation, we restrict the developer-week panel constructed in Appendix C in three steps, which together determine which developer-weeks enter the regressions and how weeks without recorded activity are treated: an active-span rule, an event-time grid, and a set of pair-level requirements. All are applied identically to treated and matched control developers.

Active Span On GitHub we observe each account's creation date, but accounts are rarely deleted: holding an account does not imply active use of the platform, and developers commonly leave without closing their accounts. To construct windows that reflect when a developer is active, we define each developer's *active span* as the interval between their first and last week with strictly positive commits, computed on the full 2020–2026 panel. Weeks outside the span are excluded.

Our *active span* can yield false positives due to natural gaps in activity that run into the end of the sample: a sporadically active developer's final weeks would be classified as inactive even if they remained on the platform. We therefore drop the final six calendar weeks of the panel after computing the active span; the trim is uniform across tools and applied to treated and control developers alike. Beyond this we impose no balanced-panel requirement: late adopters simply contribute fewer post-adoption weeks, and the number of pairs underlying each event-time coefficient is reported alongside the estimates.

Event-Time Grid Because controls are anchored one year before their matched adopter, treated and control developers cover different stretches of calendar time and can differ in data availability (for example, controls, starting a year earlier, have more post-period data). We therefore construct observations in event time: for each treated and control account, the estimation panel spans weeks −52 to +52 around its own anchor: the adoption week t_i^* for treated developers and the pseudo-adoption week t_i^p for controls.

Pair-Level Requirements Two further rules operate at the pair level. First, both members of a pair must be observed at event week −1, the reference week. Second, since our main specification divides by pre-adoption activity to obtain percent effects, pairs in which either the treated or the control developer has no activity in the outcome variable over weeks −11 to −1 are dropped. [46]

[46] This same selection is applied to the level specification as well—even though it does not divide by the individual pre-period mean—so that the two specifications are estimated on the same sample.

D.4 Event Study Specification

For every outcome we estimate the same fixed-effects event study on the matched pairs. Let Y_{it} denote the outcome for treated developer i in week t and $Y_{it}^{control}$ that of the matched control, aligned so that the control's week 0 is its pseudo-adoption week t_i^p (Appendix D.2).

Normalization To facilitate cross-developer comparison and address heterogeneity in baseline activity, we normalize the treated and the control outcome by their own pre-period means:

$$\widetilde{Y}_{it} = \frac{Y_{it}}{\bar{Y}_i^{pre}}, \quad \widetilde{Y}_{it}^{control} = \frac{Y_{it}^{control}}{\bar{Y}_i^{pre,control}}, \tag{31}$$

where $\bar{Y}_i^{pre}$ is the average of the treated developer's outcome over pre-period weeks -11 to -1, and $\bar{Y}_i^{pre,control}$ is defined analogously for the matched control. For the asynchronous agent outcomes, whose own pre-period mean is mechanically zero, the denominator is instead the developer's pre-period mean of the corresponding overall outcome.

Specification The event-study regression is:

$$\widetilde{Y}_{it} - \widetilde{Y}_{it}^{control} = \sum_{k=-11, k\neq -1}^{30} \beta_k \cdot \mathbf{1}\{t - t_i^* = k\} + \gamma_{pre} \cdot \mathbf{1}\{t < t_i^* - 11\} + \gamma_{post} \cdot \mathbf{1}\{t > t_i^* + 30\} + \alpha_i + \varepsilon_{it} \tag{32}$$

where t_i^* denotes the treatment event week for developer i, $\mathbf{1}\{\cdot\}$ is the indicator function, and α_i captures matched-pair fixed effects. The regression is estimated on the full -52 to $+52$ event grid of Appendix D.3; the indicators γ_{pre} and γ_{post} absorb the event weeks outside the -11 to $+30$ coefficient window, and the base period is $k = -1$. Standard errors are clustered at the pair level (equivalently, the treated developer). Coefficients are read as changes relative to the pre-period baseline ($\beta_k = 0.5$ indicates a 50% increase relative to the pre-period average). Because controls are observed one calendar year before their treated partner, calendar-time shocks are not differenced out; the identifying assumptions of this time-shifted design are discussed in Appendix F.

De-meaning Adjustment We re-center the estimated coefficient path on the pre-period by subtracting the average of the pre-period coefficients (with the omitted $\hat{\beta}_{-1}$ entering as zero) from every coefficient:

$$\tilde{\beta}_k = \hat{\beta}_k - \frac{1}{11} \sum_{j=-11}^{-1} \hat{\beta}_j. \tag{33}$$

This shifts the entire path by a constant, so week-to-week differences are unchanged; post-period coefficients are read relative to the average pre-adoption level (the same base-

line as the binned specification below), and the pre-period coefficients average zero by construction.

Post-Period Binned Regression To summarize treatment effects and facilitate comparison across tools, we estimate a binned specification that groups post-period weeks:

$$\begin{aligned}\widetilde{Y}_{it} - \widetilde{Y}_{it}^{control} = \quad & \beta_0 \cdot \mathbf{1}\{t - t_i^* = 0\} \\ & + \beta_{1\text{-}10} \cdot \mathbf{1}\{1 \le t - t_i^* \le 10\} \\ & + \beta_{11\text{-}20} \cdot \mathbf{1}\{11 \le t - t_i^* \le 20\} \\ & + \beta_{21\text{-}30} \cdot \mathbf{1}\{21 \le t - t_i^* \le 30\} \\ & + \gamma_{pre} \cdot \mathbf{1}\{t < t_i^* - 11\} + \gamma_{post} \cdot \mathbf{1}\{t > t_i^* + 30\} + \alpha_i + \varepsilon_{it} \end{aligned} \tag{34}$$

The omitted category is the pre-period (weeks −11 to −1), so the binned coefficients need no de-meaning: $\beta_{1\text{-}10}$ is the average treatment effect during weeks 1–10 relative to the pre-period baseline, and similarly for the other bins. Fixed effects, clustering, and sample are as in the weekly specification; the asynchronous outcomes use the same binned specification.

Pooled Specification For pooled analysis across coding agents, we stack the per-tool panels, so the unit of observation becomes the developer-by-tool pair. The *sync* pool stacks the four sync-capable tools (Claude Code, Codex, GitHub Agent, and GitHub Sync), with commits measured *outside* agent pull requests for the two async tools so that the pooled outcome captures the developers' own activity. The *async* pool stacks the two asynchronous agents (Codex and GitHub Agent) on the in-pull-request outcomes. Each developer-by-tool unit receives its own matched-pair fixed effect, so a developer who adopts several tools appears once per adoption and is treated as an independent unit (about 9% of sync-pool units and 6% of async-pool units arise from such multiple adoptions). Standard errors are clustered at the developer-by-tool level.

Winsorization and Estimation We winsorize dependent variables at the 1% level. For the matched difference specifications, the treated-minus-control difference is winsorized at both tails (1st and 99th percentiles), with the level and normalized outcomes winsorized on their own separate distributions; percentiles are computed over the full −52 to +52 event grid, after the sample rules of Appendix D.3 are applied. For the asynchronous outcomes we winsorize at the upper tail only (99th percentile), computed over positive values in event weeks 0 and later. All regressions are estimated using the `fixest` package in R.

Table OA-6: Sample Sizes for the Production-Hierarchy Regressions

	Lines Chg	Dist Files	Commits	PRs	Dist Repos	Releases
			Panel A. Matched Pairs			
Autocomplete	14,190	14,220	14,279	10,629	14,224	1,609
Sync Effect	137,727	138,613	140,118	55,220	138,693	8,912
Claude Code	19,849	19,924	20,090	9,073	19,930	1,773
GitHub Sync	67,343	67,727	68,372	31,880	67,765	5,303
OpenAI Codex	50,535	50,962	51,656	14,267	50,998	1,836
Async Effect	111,985	112,767	114,104	42,142	112,838	—
GitHub Agent	61,450	61,805	62,433	27,875	61,840	—
OpenAI Codex	50,535	50,962	51,671	14,267	50,998	—
			Panel B. Developer-Week Observations			
Autocomplete	1,287,548	1,290,189	1,295,337	973,157	1,290,587	159,985
Sync Effect	9,515,593	9,568,335	9,667,508	4,004,627	9,573,318	705,574
Claude Code	1,497,312	1,502,171	1,513,892	700,005	1,502,608	145,713
GitHub Sync	4,550,281	4,572,658	4,614,314	2,260,261	4,574,939	410,397
OpenAI Codex	3,468,000	3,493,506	3,539,302	1,044,361	3,495,771	149,464
Async Effect	7,576,979	7,623,093	7,710,028	2,996,262	7,627,467	—
GitHub Agent	4,108,979	4,129,587	4,169,672	1,951,901	4,131,696	—
OpenAI Codex	3,468,000	3,493,506	3,540,356	1,044,361	3,495,771	—

Notes: Each cell corresponds to the binned normalized event-study regression behind the same cell of Table 6 (Equation (34); weeks 21–30 bin). "Matched pairs" is the number of treated-control pairs (developer-by-tool units for the pooled rows) entering the regression after the sample rules of Appendix D.3; "Developer-week observations" is the number of pair-week cells. Pair counts differ across outcomes within a row because the positive-pre-period-mean rule is outcome-specific. The "GitHub Sync" row combines GitHub Sync adopters with the async-agent adopters whose human-authored activity enters the sync pool. Releases are not estimated for the async rows (see Table 6).

Regression Samples Table OA-6 reports, for each regression underlying the production-hierarchy results of Table 6, the number of matched pairs and the number of developer-week observations.

D.5 Heterogeneity by Pre-Period Activity

To examine how treatment effects vary with baseline developer activity, we estimate a quartile interaction specification. Each pair is assigned to a quartile $q \in \{1, 2, 3, 4\}$ of

$$\bar{M}_i^{pre} = \frac{1}{2}\left(\bar{Y}_i^{pre} + \bar{Y}_i^{pre,control}\right), \tag{35}$$

the average of the treated and control normalization denominators from Equation (31).[47] Averaging the two sides matters: if pairs were sorted on the treated developer's mean alone, developers in the top quartile would be there partly because of positive noise in that mean and would regress toward it after adoption, while their controls, not selected on this noise, would not; the resulting reversion would contaminate the quartile-specific treatment effects. Averaging makes the selection symmetric in the two sides, so mean reversion affects treated and control outcomes alike and drops out of the difference. Quartile breakpoints are the 25th, 50th, and 75th percentiles of $\bar{M}_i^{pre}$, computed within each regression sample; ties at a breakpoint join the lower group. We then interact the quartile indicators with the post-period bins of Equation (34):

$$\begin{aligned}\widetilde{Y}_{it} - \widetilde{Y}_{it}^{control} = \quad & \sum_{q=1}^{4} \sum_{b \in \mathcal{B}} \beta_{q,b} \cdot Q_q(i) \cdot \mathbf{1}\{t - t_i^* \in b\} \\ & + \gamma_{pre} \cdot \mathbf{1}\{t < t_i^* - 11\} + \gamma_{post} \cdot \mathbf{1}\{t > t_i^* + 30\} + \alpha_i + \varepsilon_{it} \end{aligned} \tag{36}$$

where $Q_q(i) = 1$ if pair i belongs to quartile q and $\mathcal{B} = \big\{\{0\}, \{1, \ldots, 10\}, \{11, \ldots, 20\}, \{21, \ldots, 30\}\big\}$. The normalized outcome is used in every specification, including the asynchronous outcomes; the omitted category is the full pre-period for all quartiles, so no de-meaning is applied; fixed effects and clustering are as above (developer-by-tool for the pools). Since the dependent variable is normalized by the pre-period mean, we report $100 \times \beta_{q,\text{21-30}}$ as the percentage treatment effect for quartile q in weeks 21–30.

D.6 Other Event-Study Specifications

The remaining empirical results in the paper use specifications beyond those above; we document each here.

Calendar-Time Effects (Figure 5) Each panel is the matched normalized event study of Equation (32), estimated on the subset of developers who adopted within the first three weeks of the tool's release (for Claude Code, its general-availability date of May 22, 2025; for the other tools, the release dates of Appendix C), together with their matched controls. Within this fixed cohort, event time maps directly onto calendar time, so the horizontal axis can be labelled by calendar date; the specification, sample rules, and inference are otherwise unchanged.

[47] For the asynchronous outcomes, where the control series is identically zero, we use $\bar{M}_i^{pre} = \bar{Y}_i^{pre}$, the developer's pre-period mean of the corresponding overall outcome, so pairs are sorted by baseline total activity.

Mechanism Outcomes (Table 7, Panel A) Panel A reports the same normalized post-period binned estimate (weeks 21–30 against the −11 to −1 baseline) applied to six mechanism outcomes. Lines per commit and files per commit are weekly averages per commit; because they are undefined in weeks without commits, these two outcomes are estimated conditional on weeks with at least one commit. Lines added, lines deleted, and branches created and deleted are weekly counts. Additions and deletions are estimated on a common sample, as are branches created and deleted: in each case we take the intersection of pairs passing the positive pre-period-mean rule for both outcomes, so the two columns describe the same developers. Branch outcomes derive from data available only from March 2023 onward; since every autocomplete adoption occurs in 2022, the autocomplete branch cells have no estimation sample and are not reported.

Repository Abandonment (Table 7, Panel B) The abandonment series is computed from the raw records of every repository that a treated developer contributed to, not from the matched panel. We use a time series rather than an event study because abandonment is a one-time, repository-level event rather than a weekly developer outcome; a developer-week event study has little power against it. A repository counts as abandoned when the last push it ever receives falls within one month of its creation. For each calendar year 2021–2026 we report the share of repositories created in that year that are abandoned; the standard error is that of the mean of the year's monthly abandonment shares. The series starts in 2021 because push timestamps are unreliable before August 2020, and 2026 covers repositories created in January through March, so that every repository is observed for at least three months beyond its first month before the July 2026 data snapshot.

Release Size and Comment Share (Table 7, Panel B) The release statistics are computed from a random draw of one million releases of repositories in our sample universe, of which 916,071 stable releases (excluding drafts and pre-releases) enter the sample. We sample because each statistic requires downloading the release's full source archive; doing this for every release is prohibitively costly in download time and storage. For each release we download the source archive, keep code files only, and classify every line as blank, comment, or code using extension-specific comment grammars. For each calendar year we report the median lines and files per release, with bootstrap standard errors from 300 release-level resamples, and the comment share (total comment lines over total lines, pooled across the year's releases), with a bootstrap standard error computed the same way. The 2026 figures cover January through March.

D.7 Aggregate Series

GitHub Aggregate Series (Figure 10) The aggregate GitHub results are descriptive monthly series (Appendix C.6); no regression is estimated. The construction of each panel is described in the note to Figure 10; the rolling averages are three-month centered, and the pre-2025 trend is a linear fit on the months through December 2024.

Marketplace Cohorts (Figure 11) The marketplace results are likewise descriptive series built from the monthly app panels of Appendix C.7; no regression is estimated. The cohort definition, usage measures, and usage-bin thresholds are given in Section 7 and the note to Figure 11. We clarify three additional details. Late-arriving incumbents, which we exclude from the cohorts, are apps entering the catalog more than three months after their store release date, whose first-seen month hence reflects catalog arrival rather than genuine entry. The cohort-usage series stops three months before the end of the sample, so that every cohort's three-month usage window is complete. All rolling averages, including those applied to the bottom-row bin shares, are three-month centered.

D.8 Production Function Calibration

We calibrate the structural parameters (θ, σ) to the empirical attenuation pattern for autocomplete, which enters the hierarchy only at layer 1 (Section 6.3); this subsection states the procedure formally. We assume θ and σ are constant across layers.

Let $\hat{g}_s$ denote the empirical productivity gain at layer s for autocomplete, in percent, and let $\hat{G}_s = 1 + \hat{g}_s/100$ denote the corresponding gross gain. We set $A = \hat{G}_1$ and recursively define the predicted gross gain at higher layers by composing the single-layer pass-through formula in (8):

$$G_s(\theta, \sigma) = \left[\theta\, G_{s-1}(\theta, \sigma)^{\rho} + (1 - \theta)\right]^{1/\rho}, \qquad \rho = \frac{\sigma - 1}{\sigma}, \qquad s = 2, \ldots, S.$$

We then choose (θ, σ) to minimize the sum of squared log differences between predicted and empirical gross gains across all six layers:

$$(\hat{\theta}, \hat{\sigma}) \in \arg \min_{\theta \in (0,1),\, \sigma > 0} \sum_{s=1}^{S} \left[\ln G_s(\theta, \sigma) - \ln \hat{G}_s\right]^2.$$

Under fixed allocation the $s = 1$ term is identically zero, because A is set to the observed lines gain; the baseline calibration is therefore equivalently a sum over layers 2 through S. This is not so under reallocation, where the layer-1 gain is itself a fitted moment (Appendix B.5).

Finally, we predict the attenuation pattern for sync and async agents by taking each tool's empirical lines-of-code effect as A and applying the same recursion with the calibrated parameters (Figure 9); Section 6.3 discusses the interpretation of these counterfactual predictions.

E Robustness Checks

In this appendix, we present the full set of robustness checks summarized in Section 5.5. Our analysis relies on observational data, so these checks are intended to support a causal interpretation of our estimates.

E.1 Robustness to Matching

Our baseline design matches each treated developer to a control developer with at least one commit and one pull request at the pseudo-adoption week, one year before the treated developer's adoption date, selecting the donor closest in pre-period commits (Appendix D.2). In this section we consider robustness to this procedure along four dimensions: the timing of the control group, the activity requirements imposed at the (pseudo-)adoption week, matching on additional observables, and the pre-period window used for matching. The results are reported in Tables OA-11 and OA-12 and Figure OA-11. We run these variants for commits and pull requests only, since retrieving the remaining outcomes for the full control pool is prohibitively costly.

Contemporaneous Controls Instead of drawing controls from one year before adoption, we re-match each treated developer to a non-adopter satisfying the same activity requirements at the adoption week itself. The results are reported in Table OA-11 and Figure OA-11: the estimates are smaller—consistent with contemporaneous controls being partially treated from private (i.e., non-observable) adoption of AI tools, the concern that motivated the lagged design—but remain large and precisely estimated.

Adoption-Week Activity Requirements In the baseline, control developers must have both a commit and a pull request in the (pseudo-)adoption week for every tool, while treated developers face this requirement only for the tools identified from Microsoft records (Appendix D.2). We re-match under uniform activity requirements, requiring a commit only, a pull request only, or both, enforced identically for every tool. [48] The results are reported in Table OA-11 and Figure OA-11 and are broadly similar: the pooled sync estimates are stable across gates, while the autocomplete estimate is somewhat sensitive to the commit-only gate.

Additional Observables We additionally require the matched control to share the treated developer's account-creation year. We do not impose this requirement in the baseline because it lowers the matching rate. The results are reported in Table OA-11 and Figure OA-11 and are very close to the baseline.

[48] As expected, the uniform commit-and-pull-request variant reproduces the paper specification exactly for the tools whose baseline already imposes both requirements.

Matching Window The baseline matches on pre-period activity over the eleven weeks before the (pseudo-)adoption week and normalizes by the same window. We condition on this relatively short window because it improves match quality and because it controls for the activity spell around adoption (the burst of engagement that coincides with adopting a new tool), which is precisely the confound our design targets; matching on a longer pre-period would instead control for a developer's longer-run activity level. We nevertheless re-match and re-normalize with pre-period windows beginning −22, −32, −42, and −52 weeks before adoption; the normalization window moves with the matching window, as does the requirement of at least one week with positive commits at or before the window's start. Table OA-12 reports the results, which are broadly similar to the baseline. The async pull-request estimates rise with the window length, but this is mechanical rather than evidence of a larger effect: the interrupted-time-series outcome is normalized by the developer's pre-period mean, which declines as the window extends into less active periods.

E.2 Robustness to Empirical Specification

Our headline estimates report the average percentage treatment effect, scaling each developer's outcome by their pre-period mean before averaging across developers. In this section we consider robustness to this specification along four dimensions: estimating in levels rather than in normalized outcomes, tightening the winsorization threshold, removing the activity-window restriction, and estimating on a balanced sample of developers active in every outcome.

Level Regressions The level event study replaces the normalized outcome of Equation (32) with the raw pair difference:

$$Y_{it} - Y_{it}^{control} = \sum_{k=-11, k\neq -1}^{30} \beta_k \cdot \mathbf{1}\{t - t_i^* = k\} + \gamma_{pre} \cdot \mathbf{1}\{t < t_i^* - 11\} + \gamma_{post} \cdot \mathbf{1}\{t > t_i^* + 30\} + \alpha_i + \varepsilon_{it} \tag{37}$$

and is otherwise identical: the same event grid, matched-pair fixed effects, clustering, and estimation sample. Because level effects vary strongly with baseline activity, we estimate this level treatment effect separately for each pre-period-mean decile—allowing the effect to differ across baseline-activity groups—then normalize each decile-specific estimate by that decile's treated pre-mean, and finally average across deciles. Figures OA-14 and OA-15 report the decile-level event studies for autocomplete and the pooled sync effect, the final row of Figure OA-12 reports their average, and Table OA-9 reports the decile estimates and their averages across the production hierarchy for the pooled sync effect. The pooled

normalized effect is very similar to the effect one would obtain from averaging the decile-specific effects, and the pattern of effects attenuating across the production hierarchy is robust across deciles. The main imperfection appears in the top two deciles of the pooled sync effect (the final two panels of Figure OA-15): treated developers' commits rise relative to their matched controls' over the weeks leading into adoption. This is precisely where matching is hardest: controls are drawn by nearest-neighbor matching on pre-period commits without replacement, and the donor pool thins out at the very top of the activity distribution, so the most active adopters receive the least comparable matches. Because level differences scale with activity, proportionally small mismatches that are invisible in the lower deciles become visible in the top ones. The lower deciles, where matches are close, show no such pre-trend.

Winsorization Our baseline winsorizes the treated-minus-control difference at the 1st and 99th percentiles. We re-estimate the event studies winsorizing at the 0.1% level instead. We choose 0.1% winsorization rather than no winsorization at all because the outcome distributions contain a few very large outliers that would otherwise entirely dominate the estimates. The results are reported in Figure OA-12 and Table OA-7: the estimates are somewhat larger, as expected when less of the thick upper tail is trimmed, but the dynamics and the patterns across tools are unchanged, and Figure OA-13 shows that the attenuation across the production hierarchy is still present under this specification.

Activity Window Our baseline censors weeks outside each developer's active span (Appendix D.3). We re-estimate the event studies without this restriction. The results are reported in Figure OA-12 and Table OA-7 and are broadly similar to the baseline; the attenuation across the production hierarchy is likewise preserved (Figure OA-13).

Balanced Sample Because activity intensity differs across outcomes, the estimation sample differs across outcomes as well: each outcome drops the pairs with no pre-period activity in that outcome, and, for example, some developers never open pull requests. This is most pronounced for releases, which are rare events shipped by only a small fraction of developers. Table OA-6 shows the resulting sample sizes, which are markedly lower for pull requests and releases. To test whether the cross-outcome comparisons are driven by these sample differences, we re-estimate all layers on a single set of developers active in every outcome: Panel A of Table OA-8 restricts to matched pairs present in all six outcome samples, and Panel B to pairs present in the five samples from lines changed to repositories touched, excluding releases, because conditioning on releases reduces the sample the most. The results are robust and the attenuation pattern across the production hierarchy is preserved. The effect sizes are somewhat smaller, as expected: conditioning

on activity in every outcome excludes less active developers, whose percentage effects are larger (Table OA-9).

E.3 Ruling Out Activity Bias with Placebos

A central concern with our estimates is *activity bias*: that the observed effects capture periods of heightened engagement that coincide with adoption rather than a true productivity effect from the tool. We address this concern with two pieces of evidence: a comparison to non-AI coding tools, and an examination of the long-term stability of the AI treatment effects.

Comparison to Non-AI Coding Tools We evaluate two additional coding tools in Figure 6 as placebo and bounding exercises.

In the left panel, we study the adoption of GitHub Pro, a paid subscription that provides advanced collaboration and compute features but no AI capabilities. GitHub Pro is of interest because willingness to pay may signal periods of heightened engagement with the platform, potentially contaminating estimates with activity bias. Consistent with this concern, Figure 6(a) shows a sharp but short-lived increase in activity for GitHub Pro adopters relative to the control group. This effect dissipates quickly, and in the long run we estimate no impact of a GitHub Pro subscription on developer productivity. This suggests focusing on the long-run estimates from our headline event studies.

In the right panel, we examine the adoption of Docker, a widely used open-source framework that allows developers to create *containers* that precisely replicate their development environment. By improving reproducibility, Docker may plausibly increase collaboration efficiency. We study Docker adoption because it is identified in a manner very similar to the adoption of Claude Code: in both cases, adoption is inferred from the appearance of a specific configuration file committed to a repository, differing only in the file's name. In the absence of any treatment effect, Docker would therefore serve as an ideal placebo. Since Docker itself likely has a positive productivity effect, however, it is more appropriately interpreted as providing an upper bound on the magnitude of coefficients we may see in our headline event studies in the absence of a real effect.

As shown in Figure 6(b), Docker adoption is associated with a large short-run increase in activity, which may reflect either temporary activity bias or the resolution of a blocking technical constraint. This effect declines rapidly, however. While the productivity gains from coding agents stabilize at levels of around 153.2%, the Docker effect stabilizes at approximately 6.6%. Hence, even if Docker had no productivity effect and this entire treatment effect were due to activity bias—an unlikely proposition—we would still infer a 146.6% increase in productivity from the adoption of coding agents.

Table OA-10 reports the same outcomes across production layers for GitHub Pro and Docker and finds no attenuation: the placebo effects are small and show no systematic decline across layers.

Long-Term Stability of the Effect To the extent that the effects in Table 5 are driven by activity bias, we would expect this bias to be short-lived: in our placebo exercises, the initial spike in activity dissipates within roughly 10 weeks (see Figure 6). It is therefore reassuring that the treatment effects from coding agents remain stable over 30 weeks, either settling down (for the async agents) or strengthening over time (for the sync agents), as can be seen by comparing the effects in columns "Weeks 11–20" and "Weeks 21–30".

E.4 External Validity: Public vs. Private Repositories

Our analysis relies on publicly observable activity to identify adoption of AI coding tools. A natural concern is that we may miss developers who use these tools exclusively on private repositories. Using internal data from GitHub on async agent usage in November 2025, we assess the extent of this undercount. We find that our public-repository-based adoption measure identifies 24.2% of all async agent users—the remainder use the tool only on private repositories. However, for the developers we do identify as adopters, we observe 72.9% of their usage activity publicly; the remaining activity occurs on private repositories we do not have access to. In other words, our count of adopters is an undercount, which is why we do not emphasize adoption numbers. But for the adopters in our sample, their public activity captures the large majority of their usage, suggesting that our productivity estimates are based on a representative picture of these developers' workflows.

A second piece of evidence comes from repository identifiers. GitHub assigns each repository a numeric `id` at creation, and these identifiers are incremental, so the largest `id` assigned as of a given date counts every repository ever created on the platform, private ones included. Enumerating the repositories we can retrieve through the public API gives the corresponding count of public repositories, and comparing the two implies that 38% of all repositories ever created on GitHub are public. Because deleted repositories retain their identifiers and so remain in the denominator, this figure is a lower bound on the public share. Public activity therefore accounts for a substantial share of the repositories on the platform, so the data we observe are far from a narrow slice of developer work.

E.5 Robustness to Activity Deletion

Because we collect data through API calls, we observe only the activity still available on GitHub at collection time. Coding activity can disappear from this record when the account that produced it is deactivated or the repository that hosts it is deleted or made

private.

One alternative is GHArchive, which records daily snapshots of the public GitHub event stream going back more than a decade and so preserves activity that has since been removed. Our exploration of these data, however, points to quality problems in recent years. Figure OA-18 reports the number of push events recorded in GHArchive over time and shows clear instability after mid-2024 and a pronounced decline in 2025.[49] For this reason we do not use GHArchive.

To gauge how much deletion matters, we instead measure it directly: we download the full set of users and repositories on GitHub at two dates ten days apart and track how many become unavailable between the two. On an annualized basis, 0.4% of users deactivate their accounts and 5.8% of repositories are deleted. These figures likely overstate the amount of activity we lose, since deleted accounts and repositories tend to be less active than those that persist.

[49] This is likely attributable to GitHub's reduction of the data retention window for the Events API, Atom feed, `/timeline`, and `/dashboard-feed` from 90 days to 30 days, announced in November 2024 and effective January 30, 2025 (GitHub Changelog). GHArchive users have separately documented a sharp drop in recorded events beginning in May 2025 (GHArchive issue #310).

F Time-Shifted Event Studies: Identifying Assumptions

Because AI coding tools may be adopted without leaving a trace in public repositories, a contemporaneous control group risks contamination by private adoption. Same-year non-adopters are also likely selected on their future trajectory: developers who forgo a widely adopted tool are disproportionately those winding down active development. Our main design therefore compares each treated developer to a matched control observed one year earlier (Section D). This appendix states the assumptions under which this time-shifted event study identifies the treatment effect.

Let t_i^* denote treated developer i's adoption week and $c(i)$ the matched control, whose pseudo-adoption week is $t_i^p = t_i^* - L$, where L is one year. Write $\widetilde{Y}_{j,t} = Y_{j,t}/\bar{Y}_j^{pre}$ for the normalized outcome that Equation (32) uses, and suppose it follows

$$\widetilde{Y}_{j,t} \;=\; \mu_j + g_j\, t + \lambda(t) + B(t - e_j) + \tau_{t-e_j}\, D_{j,t} + \varepsilon_{j,t},$$

where μ_j is a developer fixed effect, g_j a developer-specific linear trend, $\lambda(t)$ a common calendar-time component, and τ_k the treatment effect at event time k. We state the components in normalized units: a common λ or B then means a shock proportional to a developer's scale, and the cancellations below hold for every outcome, not only those where matching equalizes the pair's pre-period means. The term $B(t - e_j)$ is a profile in time relative to j's (pseudo-)adoption date e_j ($e_i = t_i^*$, $e_{c(i)} = t_i^p$): it captures "activity bias," i.e. selection on activity around the event date, such as mean reversion after the burst of activity that triggers adoption. Our estimator regresses the time-shifted difference $\Delta_{i,t} = \widetilde{Y}_{i,t} - \widetilde{Y}_{c(i),\,t-L}$ (the dependent variable $\widetilde{Y}_{it} - \widetilde{Y}_{it}^{control}$ of Equation (32), written in calendar time) on event-time indicators with pair fixed effects. Substituting,

$$\Delta_{i,t} \;=\; \underbrace{(\mu_i - \mu_{c(i)}) + g_{c(i)}L}_{\text{absorbed by } \alpha_i} \;+\; \underbrace{(g_i - g_{c(i)})\, t + \big[\lambda(t) - \lambda(t-L)\big]}_{\equiv\, b_i(t)} \;+\; \tau_{t-t_i^*} D_{i,t} + \tilde{\varepsilon}_{i,t}.$$

Since treated and control sit at the same event time $k = t - t_i^*$, the profile $B(k)$ differences out exactly—provided it is common to both groups, which our matching targets by requiring controls to be active in the pseudo-adoption week and pairing on pre-period activity.[50] What remains is $b_i(t)$: in a balanced panel, $\operatorname{plim} \widehat{\beta}_k = \tau_k + \mathbb{E}[b_i(t_i^* + k)] - \mathbb{E}[b_i(t_i^* - 1)]$, so the design is unbiased when $\mathbb{E}[b_i(t_i^* + k)]$ is constant in k—an expectation the pre-period coefficients reveal for $k < 0$. Because each pair is compared only to its own pre-period

[50] Also recall that our estimates are robust to enforcing identical adoption-week activity requirements on both groups (Appendix E.1).

and controls remain untreated, the "forbidden comparisons" of staggered two-way fixed-effects estimators do not arise (Goodman-Bacon, 2021; Sun and Abraham, 2021). We next consider the ways in which $b_i(t)$, or the profile $B(\cdot)$, may fail to difference out, and organize them by what the pre-period coefficients would reveal: (i) cases that leave the estimator unbiased; (ii) cases that bias it but that the pre-trend test detects; and (iii) cases that bias it while leaving the pre-period undisturbed. Only the third threatens the design.

No bias. Annual seasonality, $\lambda(t) = \lambda(t - L)$, drops out of $b_i(t)$ exactly—the reason we shift by precisely one year. A common linear trend, $\lambda(t) = ct$, leaves $\lambda(t) - \lambda(t - L) = cL$, absorbed by the pair fixed effect.

Bias detected by the pre-trend test. Because adoption is staggered over many months, shocks on fixed calendar dates typically fall after adoption for early adopters but within the eleven-week pre-period of later ones, whose pre-period coefficients then move: calendar-time variation in $b_i(t)$ generally loads onto the pre-period, and the test rejects in the infinite-data limit. This covers an accelerating common trend ($\lambda(t) = ct^2$, so $b_i(t)$ is linear in t), aggregate shocks at fixed calendar dates, systematically different treated and control trends ($\mathbb{E}[g_i - g_{c(i)}] \neq 0$), a slope break in the common trend, and anticipation of adoption. The qualification is that the variation must fall within the pre-period of at least some adopters; a shock arriving only *after* all adoption dates (after 2025-12-22 for the tools launched in 2025; see Table OA-3) escapes the test.

Undetectable bias. The pre-trend test is silent about components of $b_i(t)$, or asymmetries in $B(\cdot)$, that emerge only at or after adoption; the identifying assumption is that there are none. Four cases are worth noting. First, a late aggregate shock occurring after all adoptions.[51] Second, matched controls may adopt an AI tool during the weeks serving as the treated developer's post-period, so that their treatment effect is subtracted from $\Delta_{i,t}$; this attenuates $\widehat{\beta}_k$ toward zero—making our estimates conservative—and is limited by drawing controls from the year before adoption, when these tools were less available. Third, developers may adopt at the start of a new project or burst of work, so that treated activity would have risen at t_i^* even absent treatment—an asymmetry in $B(\cdot)$ arising exactly at $k = 0$. The non-AI placebo tools in Figure 6 show the footprint of such residual selection: a sharp spike at adoption that decays within weeks, far from the persistent effects we estimate for AI tools. Finally, any shock to treated developers coinciding with adoption itself is indistinguishable from a treatment effect—the classic violation of counterfactual parallel trends, to which no difference-in-differences design is robust.

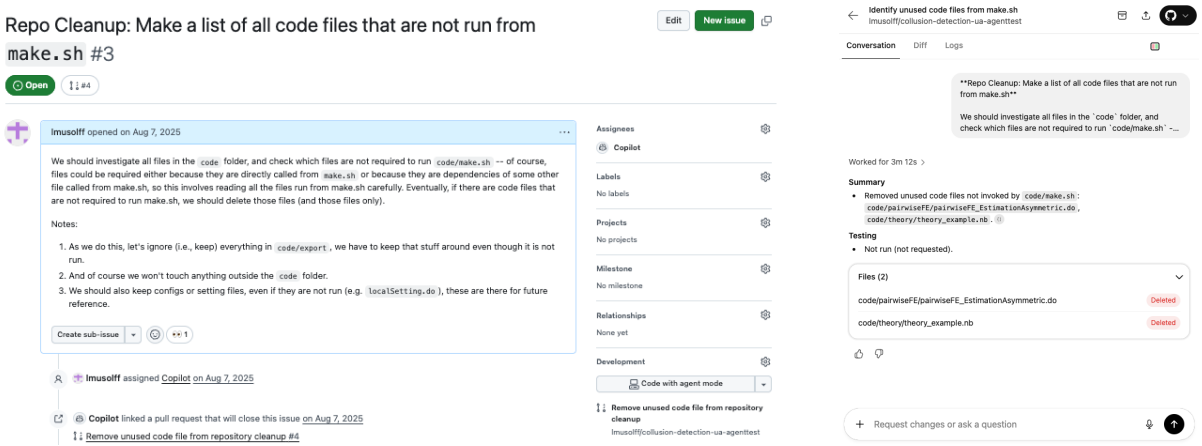
[51] Calendar-week fixed effects—separately identified from event time because adoption is staggered—would absorb any fixed-date shock. However, if private adoption is widespread in the control group, it likely also occurs among treated developers before their *public* adoption, so this repair is ultimately fruitless.

In sum, relative to a standard difference-in-differences, the time-shifted design trades robustness to aggregate shocks occurring after the last-observed adoption for robustness to contamination and to selection of non-adopters on future engagement, while relying on the same fundamental requirement of counterfactual parallel trends.

G Additional Figures and Tables

This appendix collects supplementary figures and tables referenced throughout the paper.

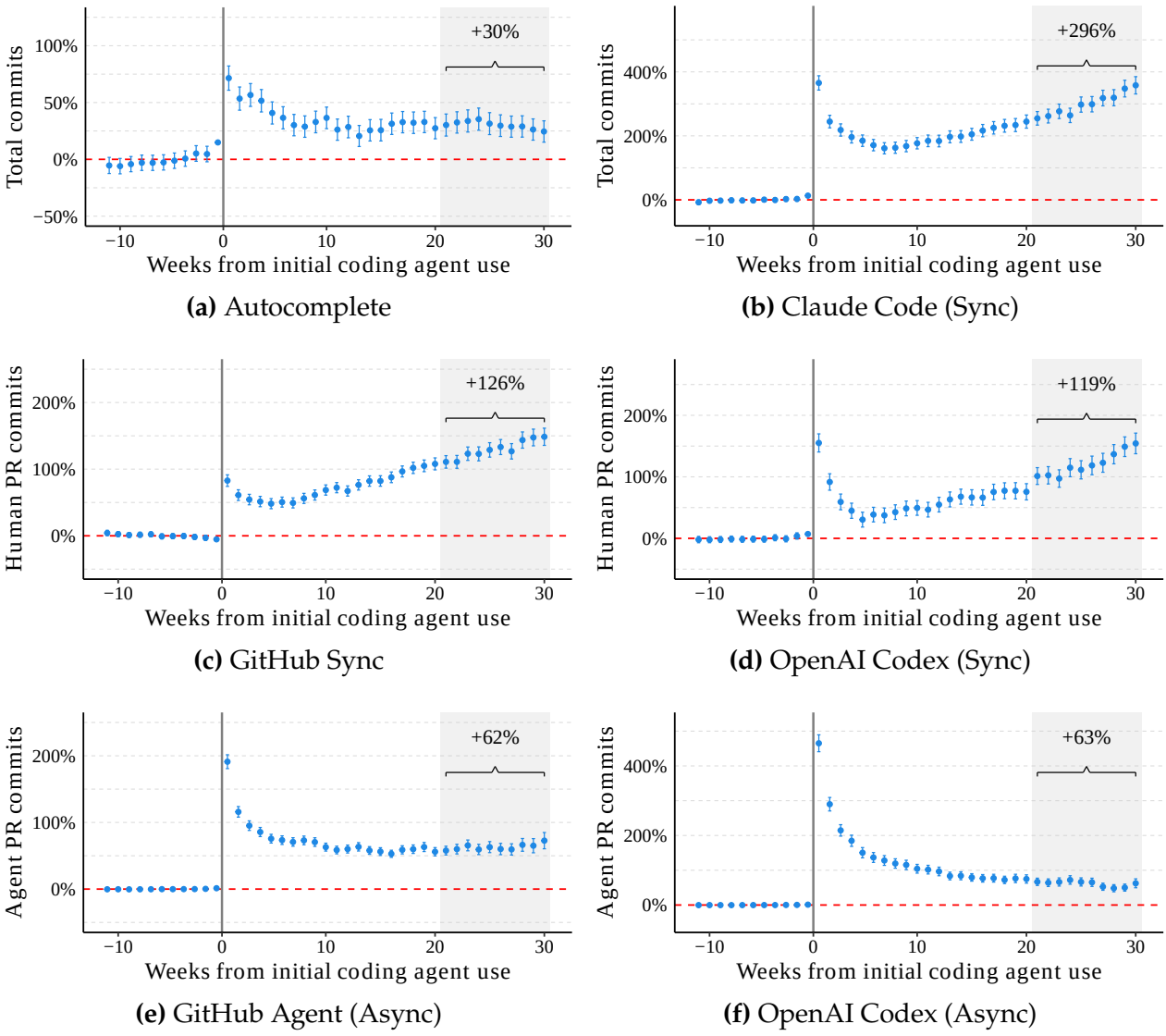
Figure OA-3: Assigning Tasks to Async Coding Agents



(a) Issue Assigned to Copilot **(b)** Codex Interface

Notes: This figure illustrates how developers can assign tasks to async coding agents. On the left-hand side, a developer has assigned an issue to the GitHub Agent directly on GitHub.com; on the right-hand side, a developer has assigned a task to OpenAI Codex on OpenAI's website.

Figure OA-4: Task-Level Productivity Effect by AI Coding Tool



(a) Autocomplete

(b) Claude Code (Sync)

(c) GitHub Sync

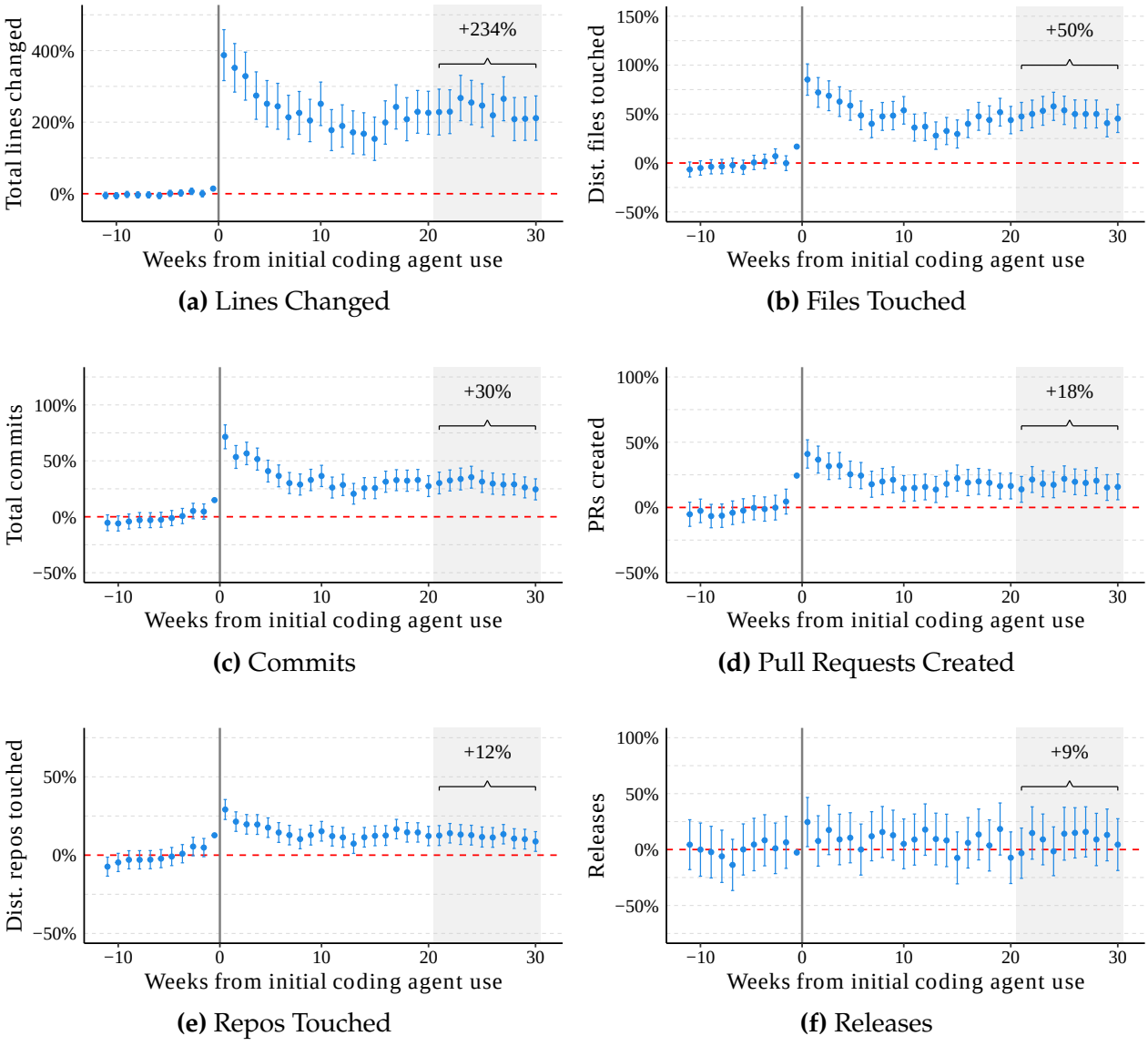
(d) OpenAI Codex (Sync)

(e) GitHub Agent (Async)

(f) OpenAI Codex (Async)

Notes: This figure presents per-tool matched event-study estimates of the effect of adopting each AI coding tool on commits, normalized by each developer's pre-period mean. The top row shows autocomplete and Claude Code; the middle row shows GitHub Sync and OpenAI Codex sync effects (human-authored commits); the bottom row shows GitHub Agent and OpenAI Codex async effects (agent-authored commits). For async tools, the outcome is zero by definition before adoption, so effects are estimated using an interrupted time series. The shaded region marks weeks 21–30 after adoption, with the annotation reporting the average effect over that window. The first post-adoption week is omitted as transitory effects distort the scale; the omitted coefficients are 102.4%, 659.1%, 102.3%, 247.0%, 596.6%, and 1094.8% for the six panels in order.

Figure OA-5: Productivity Effects Across the Production Hierarchy (Autocomplete)



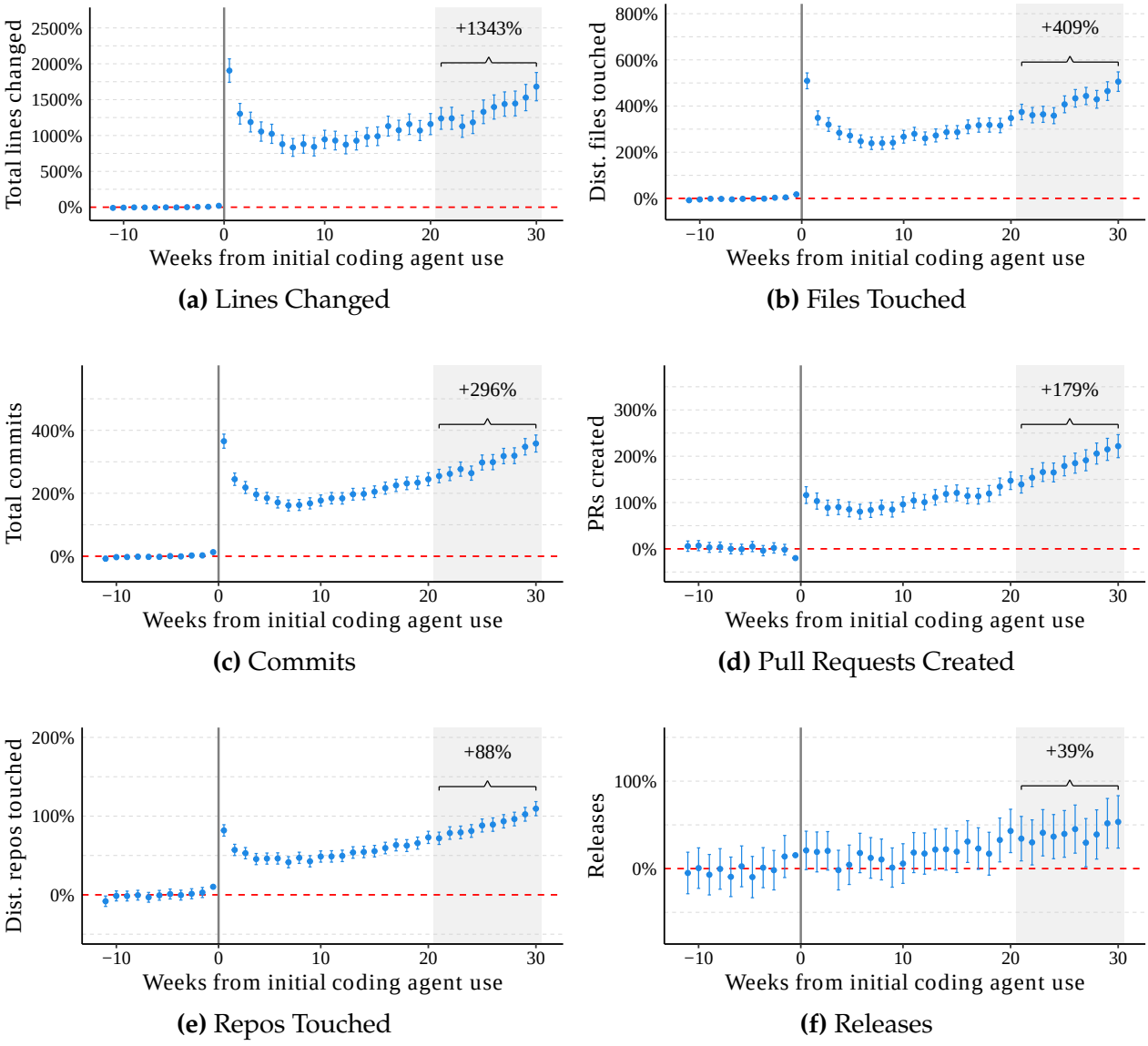
(a) Lines Changed **(b)** Files Touched

(c) Commits **(d)** Pull Requests Created

(e) Repos Touched **(f)** Releases

Notes: This figure presents matched event-study estimates of the effect of GitHub Autocomplete adoption across production layers, replicating Figure 7 for the individual tool. Outcomes are normalized by each developer's pre-period mean. The first post-adoption week is omitted as transitory effects distort the scale; the omitted coefficients are 656.6%, 127.5%, 102.4%, 73.2%, 31.0%, and 16.8% for Panels OA-5(a)–OA-5(f), respectively.

Figure OA-6: Productivity Effects Across the Production Hierarchy (Claude Code)



(a) Lines Changed

(b) Files Touched

(c) Commits

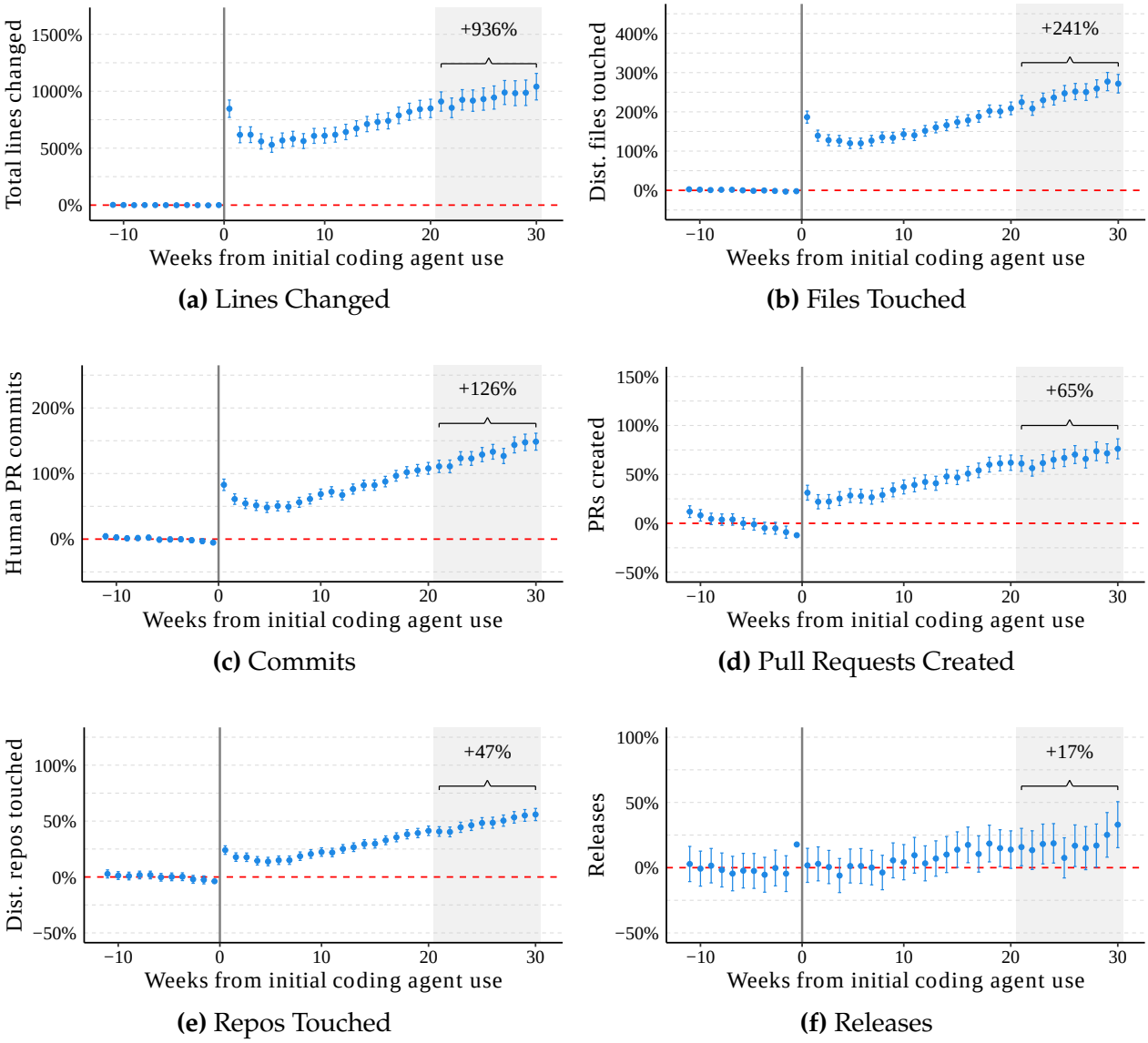
(d) Pull Requests Created

(e) Repos Touched

(f) Releases

Notes: This figure presents matched event-study estimates of the effect of Claude Code adoption across production layers, replicating Figure 7 for the individual tool. Outcomes are normalized by each developer's pre-period mean. The first post-adoption week is omitted as transitory effects distort the scale; the omitted coefficients are 4493.5%, 1133.4%, 659.1%, 53.2%, 101.6%, and 48.8% for Panels OA-6(a)–OA-6(f), respectively.

Figure OA-7: Productivity Effects Across the Production Hierarchy (GitHub Sync, Pooled)



(a) Lines Changed

(b) Files Touched

(c) Commits

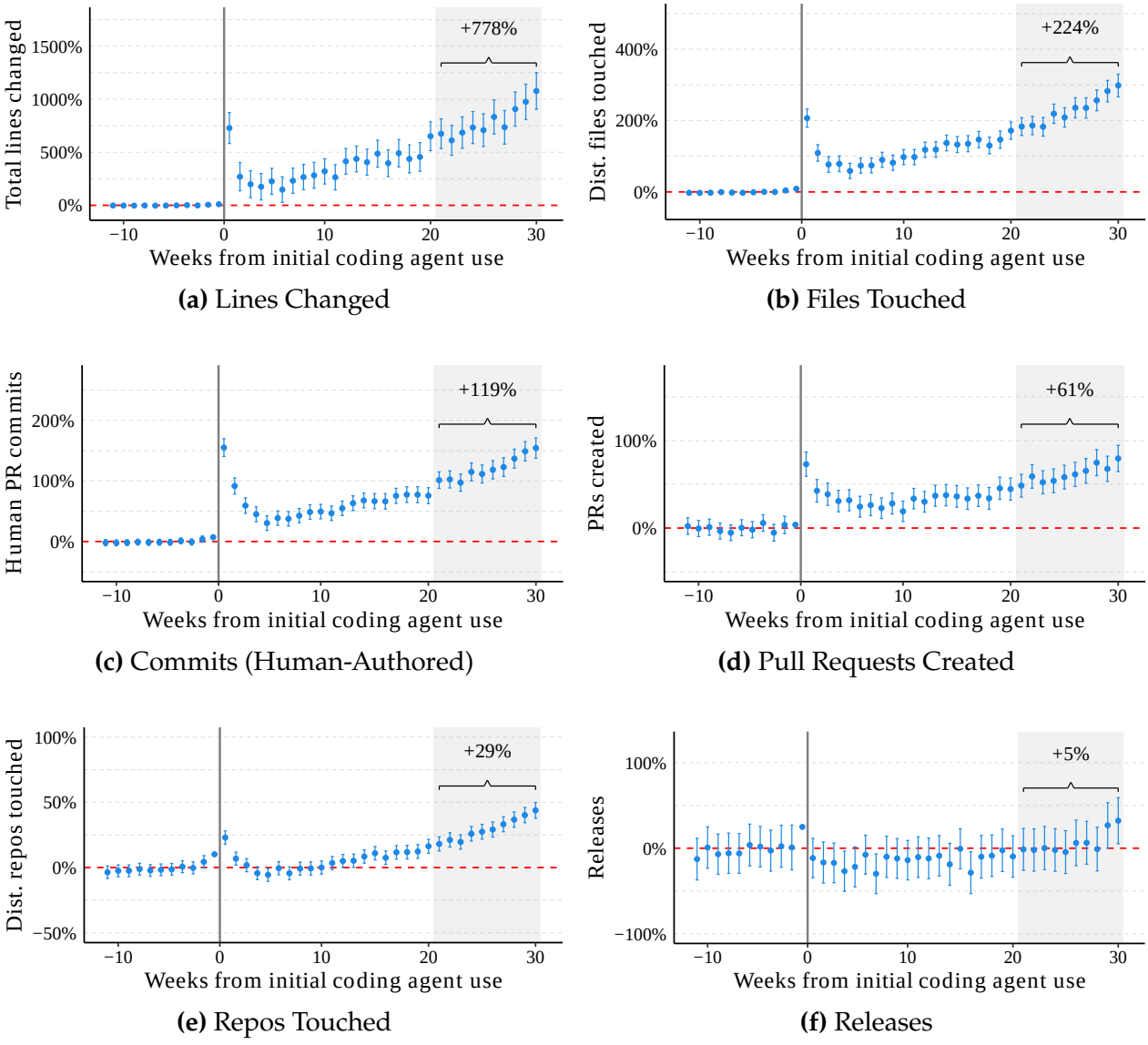
(d) Pull Requests Created

(e) Repos Touched

(f) Releases

Notes: This figure presents matched event-study estimates of the effect of GitHub Sync adoption across production layers, replicating Figure 7 for the individual tool. Outcomes are normalized by each developer's pre-period mean. The first post-adoption week is omitted as transitory effects distort the scale; the omitted coefficients are 1231.3%, 229.6%, 102.3%, −58.4%, −2.2%, and 4.9% for Panels OA-7(a)–OA-7(f), respectively.

Figure OA-8: Productivity Effects Across the Production Hierarchy (Codex Sync)



(a) Lines Changed

(b) Files Touched

(c) Commits (Human-Authored)

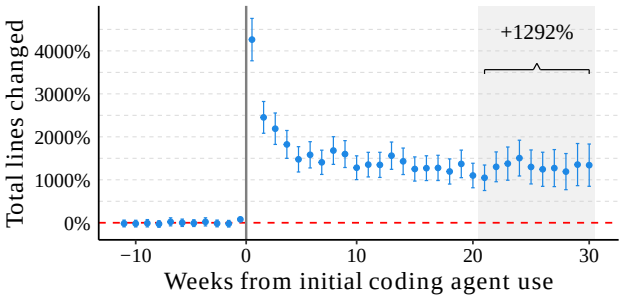
(d) Pull Requests Created

(e) Repos Touched

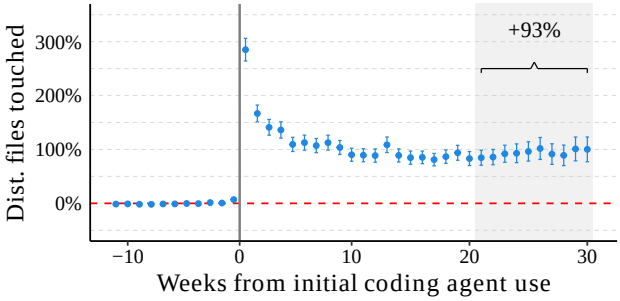
(f) Releases

Notes: This figure presents matched event-study estimates of the sync-component effect of OpenAI Codex adoption across production layers, replicating Figure 7 for the individual tool. Outcomes are normalized by each developer's pre-period mean. The first post-adoption week is omitted as transitory effects distort the scale; the omitted coefficients are 1131.3%, 322.8%, 247.0%, −16.8%, 7.4%, and 3.8% for Panels OA-8(a)–OA-8(f), respectively.

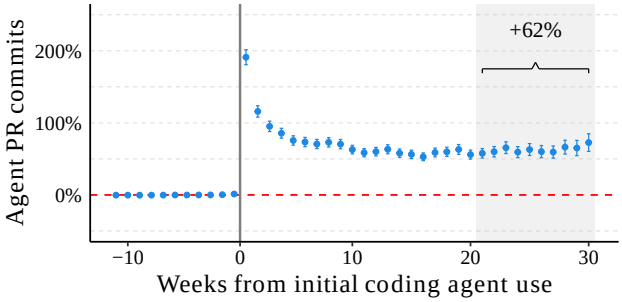
Figure OA-9: Productivity Effects Across the Production Hierarchy (GitHub Agent Async)



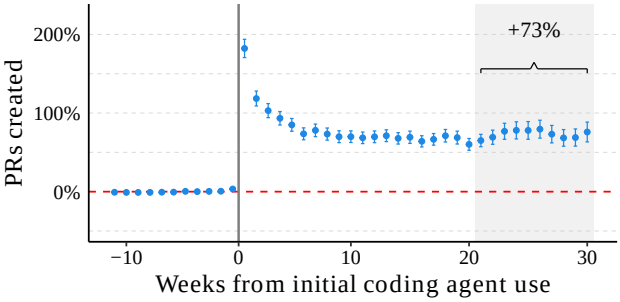
(a) Lines Changed



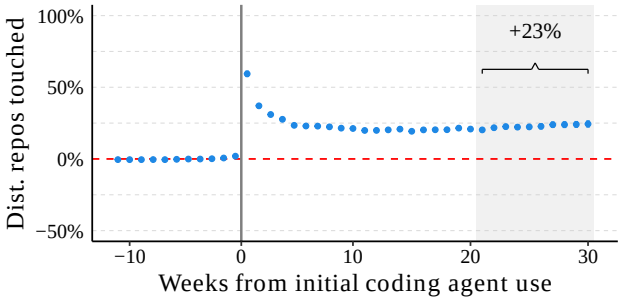
(b) Files Touched



(c) Commits (Agent-Authored)



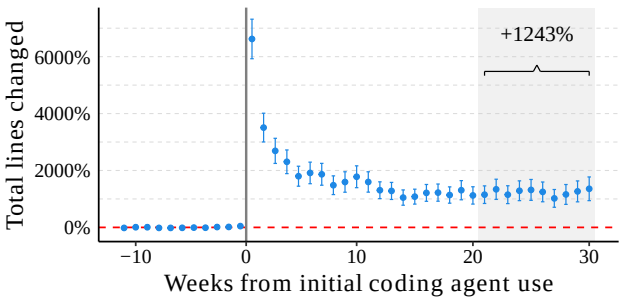
(d) Pull Requests Created



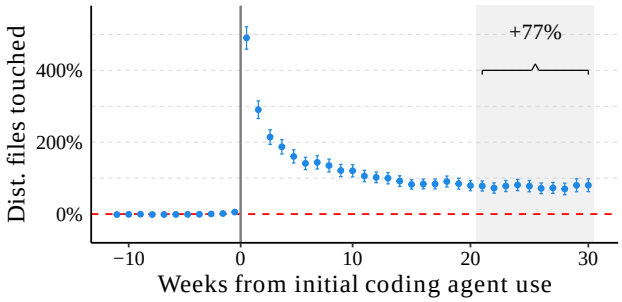
(e) Repos Touched

Notes: This figure presents interrupted time-series estimates of the async-component effect of GitHub Agent adoption across production layers, replicating Figure 7 for agent-authored output. Outcomes are normalized by each developer's pre-period mean. The first post-adoption week is omitted as transitory effects distort the scale; the omitted coefficients are 19,947.6%, 1133.3%, 596.6%, 687.2%, and 304.6% for Panels OA-9(a)–OA-9(e), respectively. The releases outcome is omitted because the release outcome is measured at the repository level and cannot be split into agent- and human-authored components.

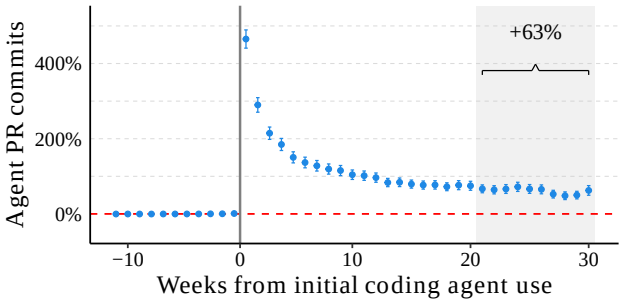
Figure OA-10: Productivity Effects Across the Production Hierarchy (Codex Async)



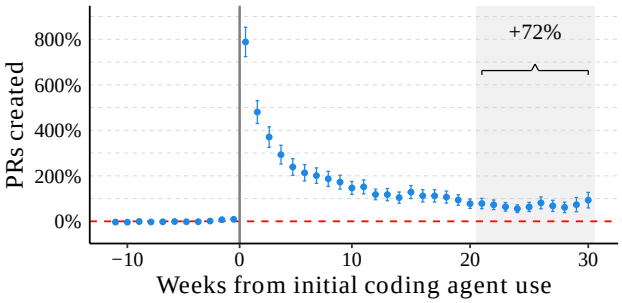
(a) Lines Changed



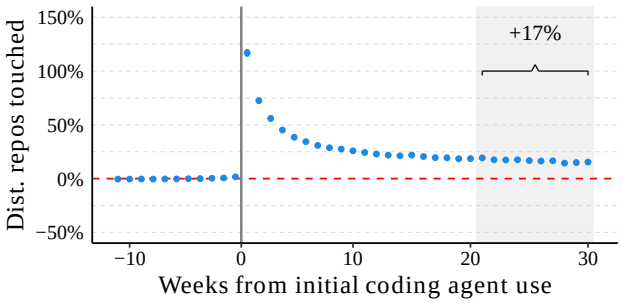
(b) Files Touched



(c) Commits (Agent-Authored)



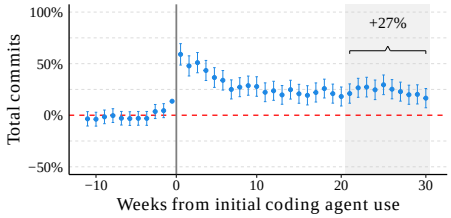
(d) Pull Requests Created



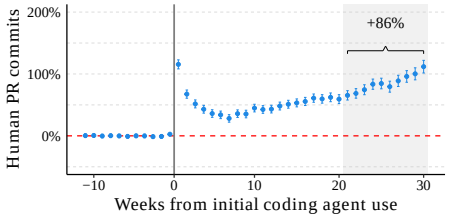
(e) Repos Touched

Notes: This figure presents interrupted time-series estimates of the async-component effect of OpenAI Codex adoption across production layers, replicating Figure 7 for agent-authored output. Outcomes are normalized by each developer's pre-period mean. The first post-adoption week is omitted as transitory effects distort the scale; the omitted coefficients are 22,037.2%, 1532.2%, 1094.8%, 1807.1%, and 495.6% for Panels OA-10(a)–OA-10(e), respectively. The releases outcome is omitted because the release outcome is measured at the repository level and cannot be split into agent- and human-authored components.

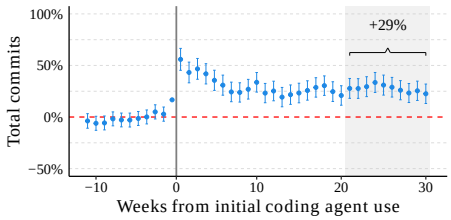
Figure OA-11: Robustness to Matching Specification



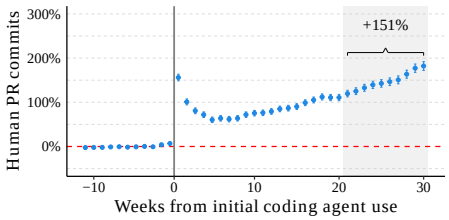
(a) Contemporaneous control: Autocomplete



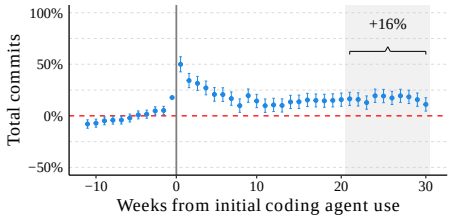
(b) Contemporaneous control: Pooled Sync



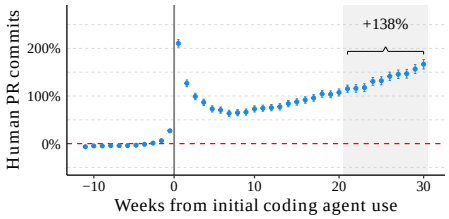
(c) Same account-creation year: Autocomplete



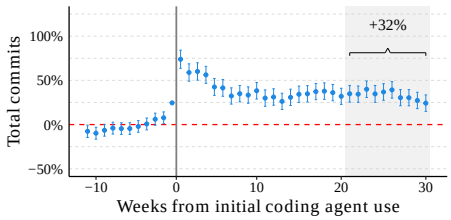
(d) Same account-creation year: Pooled Sync



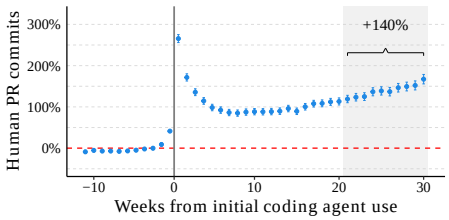
(e) Uniform commit requirement: Autocomplete



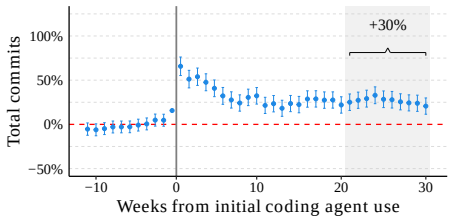
(f) Uniform commit requirement: Pooled Sync



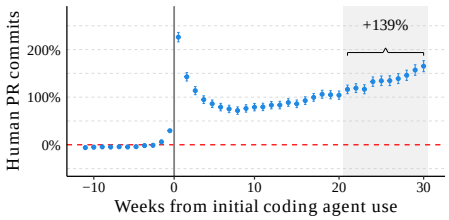
(g) Uniform PR requirement: Autocomplete



(h) Uniform PR requirement: Pooled Sync



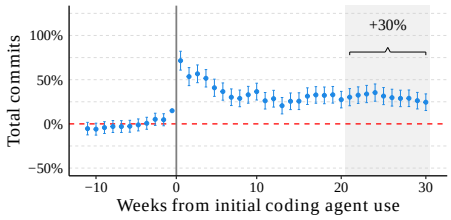
(i) Uniform commit+PR requirement: Autocomplete



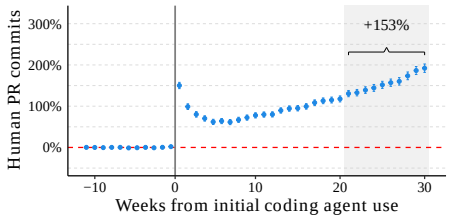
(j) Uniform commit+PR requirement: Pooled Sync

Notes: This figure presents matched event-study estimates of the effect of adopting AI coding tools on commits under alternative matching specifications; each row is one matching algorithm, with autocomplete in the left column and the pooled sync effect in the right column. Outcomes are normalized by each developer's pre-period mean. The shaded region marks weeks 21–30 after adoption, with the annotation reporting the average effect over that window. The corresponding estimates appear in Table OA-11.

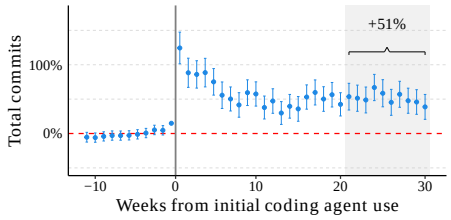
Figure OA-12: Robustness to Estimation Specification



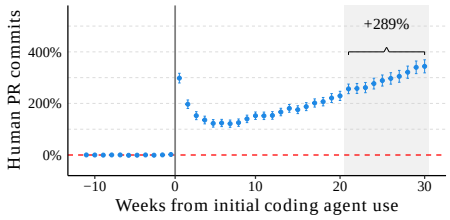
(a) Paper default: Autocomplete



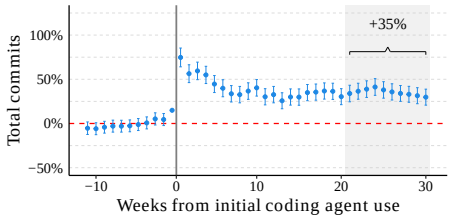
(b) Paper default: Pooled Sync



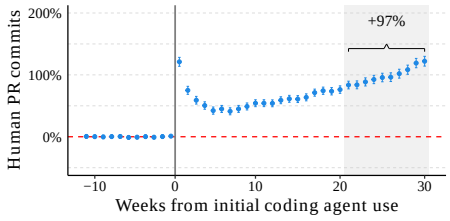
(c) 0.1% winsorization: Autocomplete



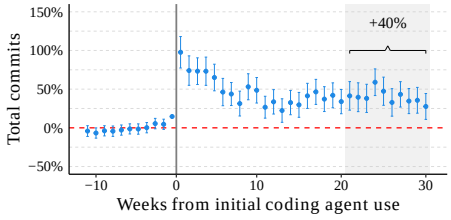
(d) 0.1% winsorization: Pooled Sync



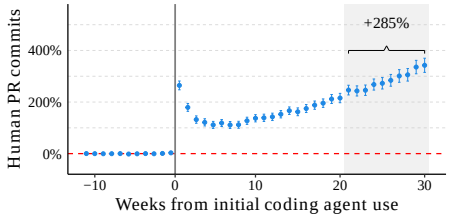
(e) No activity window: Autocomplete



(f) No activity window: Pooled Sync



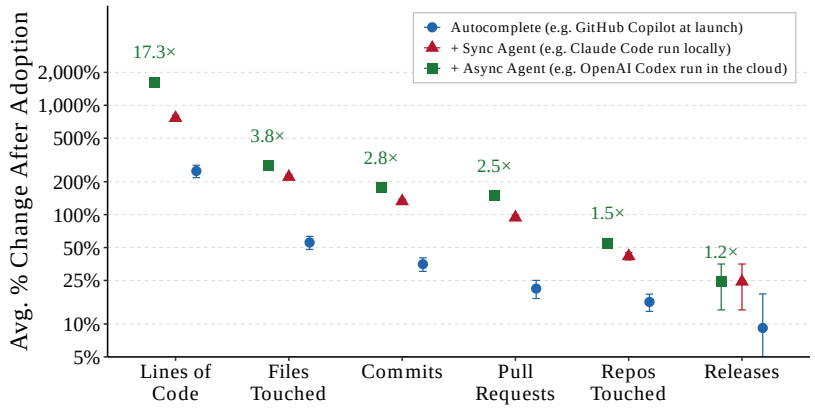
(g) Average effect: Autocomplete



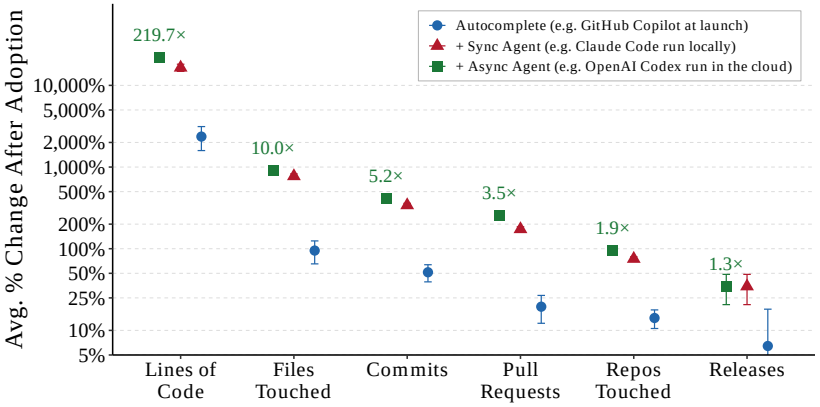
(h) Average effect: Pooled Sync

Notes: This figure presents matched event-study estimates of the effect of adopting AI coding tools on commits under the alternative estimation specifications of Appendix E.2, with autocomplete in the left column and the pooled sync effect in the right column. The first row reproduces the paper specification. The second row winsorizes at the 0.1% level rather than 1%. The third row removes the active-span restriction, retaining weeks outside each developer's active span (Appendix D.3). The final row estimates the event studies in levels separately within each decile of pre-period activity and reports the average of the ten decile-level event studies of Figures OA-14 and OA-15, weighting deciles equally. In the first three rows outcomes are normalized by each developer's pre-period mean; in the final row each decile estimate is normalized by its decile's treated pre-period mean. The shaded region marks weeks 21–30 after adoption, with the annotation reporting the average effect over that window. The estimates for the second and third rows appear in Table OA-7; the decile estimates underlying the final row are reported for the pooled sync effect in Table OA-9.

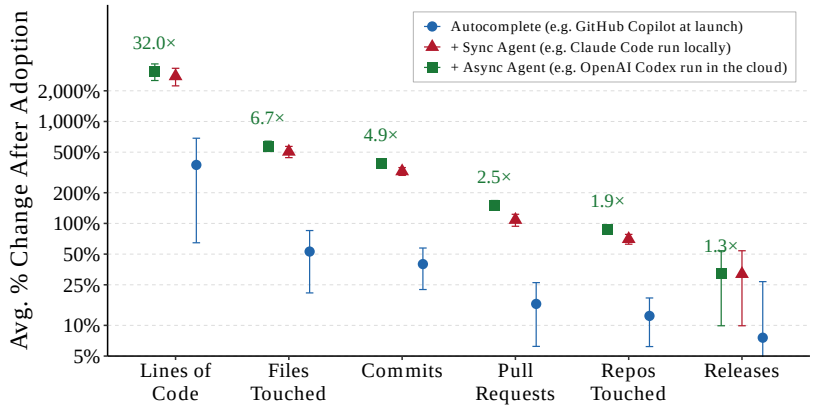
Figure OA-13: Robustness of Attenuation to Estimation Specification



(a) No activity window



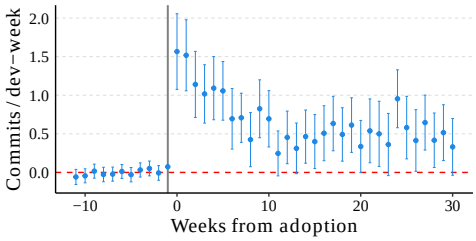
(b) 0.1% winsorization



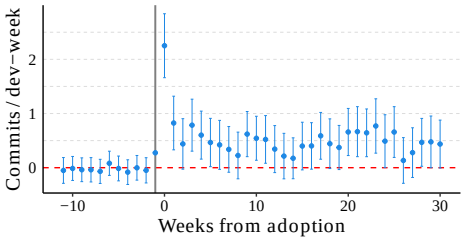
(c) Average level effect

Notes: Each panel reproduces the event-study coefficient overview of Figure 1 under one estimation specification of Appendix E.2. Panel (a) removes the active-span restriction, retaining weeks outside each developer's active span (Appendix D.3). Panel (b) winsorizes at the 0.1% level rather than 1%. Panel (c) reports the average of the ten decile-level event studies of Figures OA-14 and OA-15, weighting deciles equally. The corresponding weeks 21–30 estimates appear in Table OA-7.

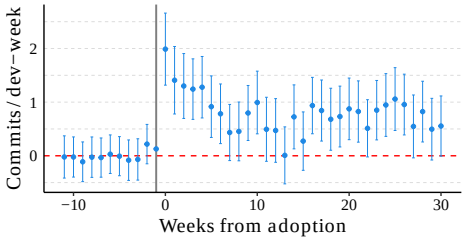
Figure OA-14: Level-Specification Event Studies by Pre-Period Decile: Autocomplete



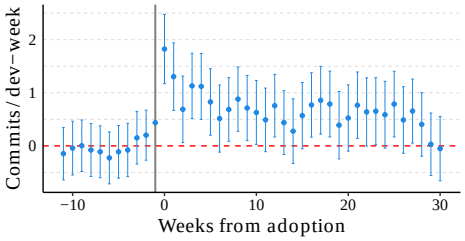
(a) Decile 1



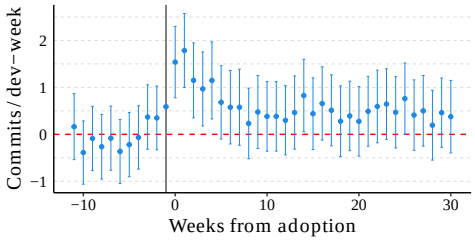
(b) Decile 2



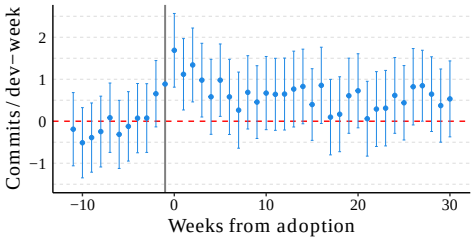
(c) Decile 3



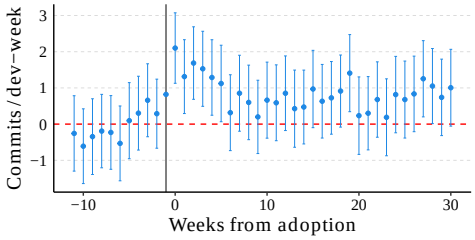
(d) Decile 4



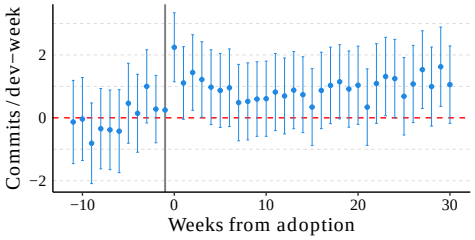
(e) Decile 5



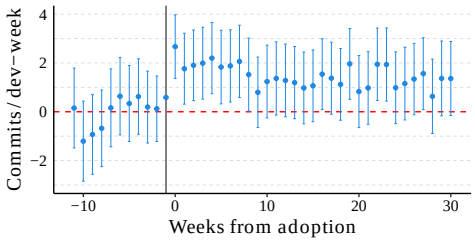
(f) Decile 6



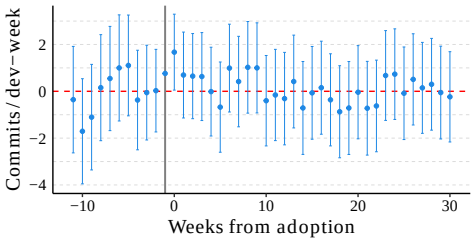
(g) Decile 7



(h) Decile 8



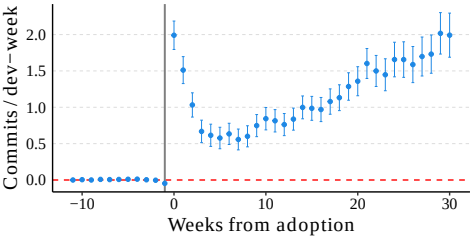
(i) Decile 9



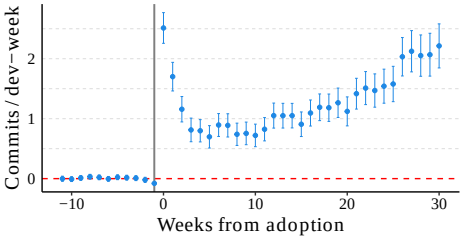
(j) Decile 10

Notes: Each panel reports the level-specification event study (Equation (37)) estimated separately within one decile of the pre-period activity distribution, for Autocomplete adopters and their matched controls.

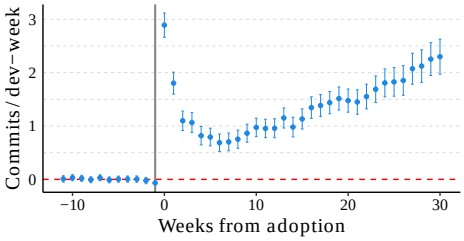
Figure OA-15: Level-Specification Event Studies by Pre-Period Decile: Pooled Sync



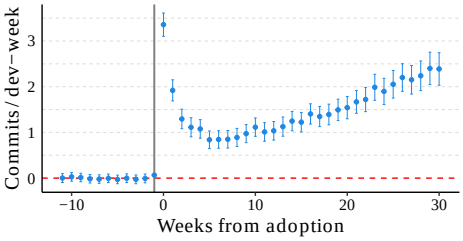
(a) Decile 1



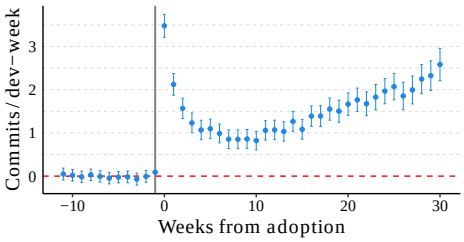
(b) Decile 2



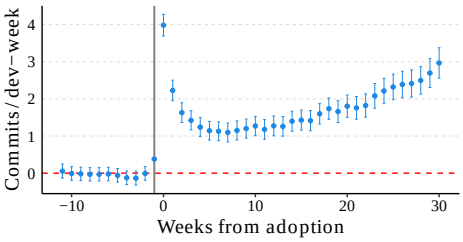
(c) Decile 3



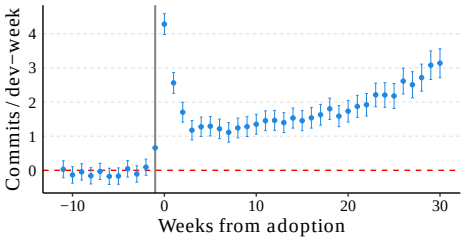
(d) Decile 4



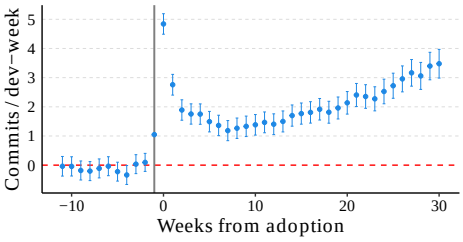
(e) Decile 5



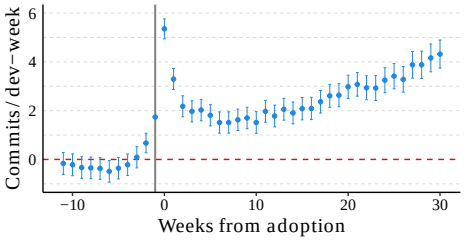
(f) Decile 6



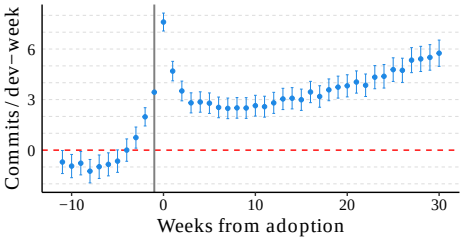
(g) Decile 7



(h) Decile 8



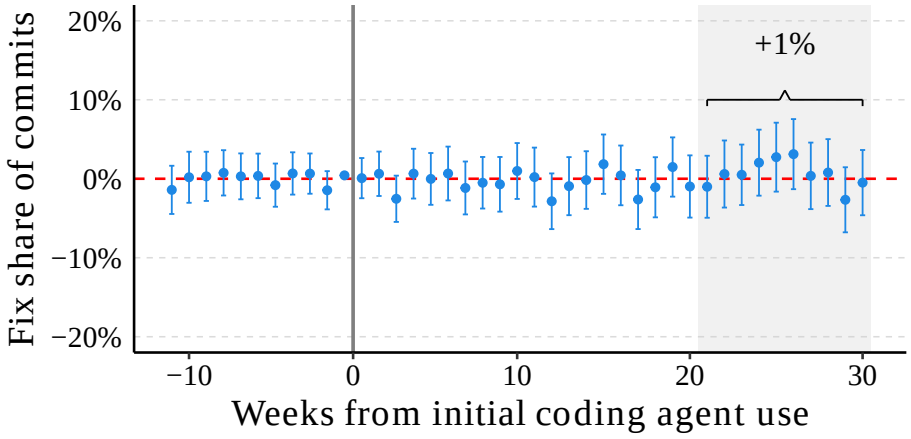
(i) Decile 9



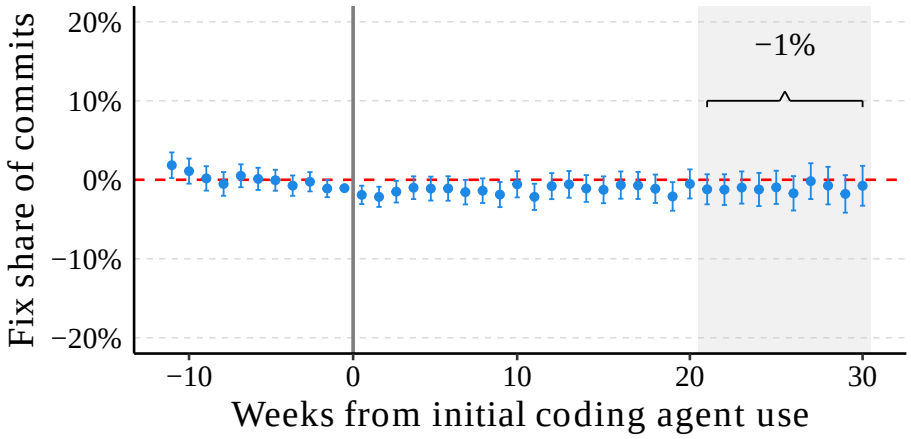
(j) Decile 10

Notes: Each panel reports the level-specification event study (Equation (37)) estimated separately within one decile of the pre-period activity distribution, for Pooled Sync adopters and their matched controls. The decile estimates are reported in Table OA-9.

Figure OA-16: Bug-Fix Share of Commits



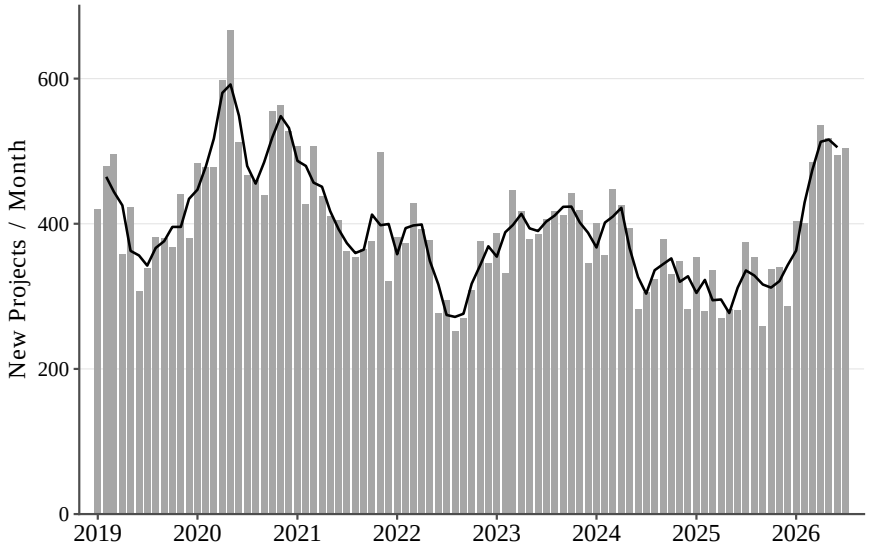
(a) Autocomplete



(b) Pooled Sync

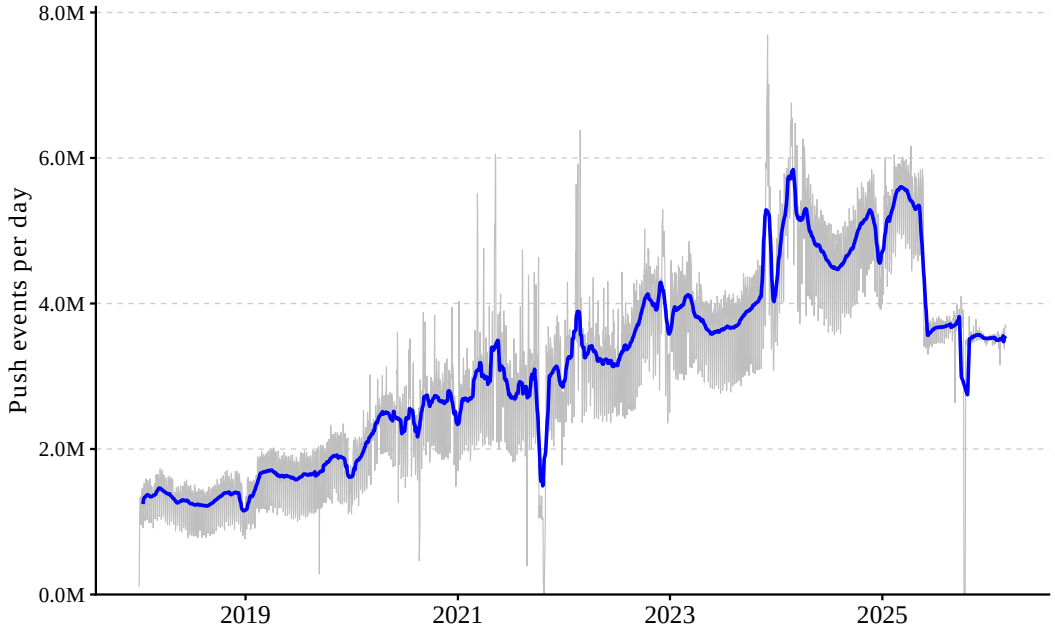
Notes: This figure presents matched event-study estimates of the treated-minus-control difference in the share of bug-fix commits among labelled commits—commits whose message subject line matches a bug-fix or a feature pattern—in percentage points, de-meaned over the pre-period with the first post-adoption week omitted. The shaded region marks weeks 21–30 after adoption, with the annotation reporting the average over that window. A commit matching both patterns counts once in the numerator and twice in the denominator (0.18% of commits), biasing the share slightly downward.

Figure OA-17: New Apps on SourceForge



Notes: Monthly count of newly created SourceForge projects from 2019 through mid-2026. Bars are raw monthly values; the solid black line is a 3-month rolling average. New projects drift down through 2025 and increase slightly at the end of the sample; compared with the iOS App Store and the Chrome Web Store (Figure 11), the acceleration around the rise of agentic coding tools is small. SourceForge's download counts are too small to support a meaningful analysis of usage, so we report entry only.

Figure OA-18: Daily Push Events Recorded in GHArchive



Notes: Daily counts of push events recorded in the GHArchive public event stream. The grey series is the raw daily count; the blue line is a 28-day centered moving average.

Table OA-7: Attenuation of Productivity Effects by Estimation Specification

	Outcomes (Weeks 21–30, %)					
	Lines Chg	Files Touched	Commits	PRs	Repos Touched	Releases
			Panel A. No Activity Window			
Autocomplete	250.5	55.7	35.3	21.1	15.9	9.2
	(16.5)	(3.9)	(2.5)	(2.0)	(1.5)	(4.9)
Sync Effect	514.4	165.7	97.2	72.9	25.8	15.2
	(17.5)	(3.6)	(2.1)	(1.8)	(0.7)	(2.7)
Claude Code	1200.3	363.9	263.6	168.2	76.2	36.7
	(45.2)	(10.4)	(6.8)	(6.0)	(2.0)	(6.5)
GitHub Sync	610.7	158.7	83.3	56.0	26.7	13.9
	(21.7)	(4.7)	(2.7)	(2.0)	(1.0)	(3.5)
OpenAI Codex	93.6	89.2	44.5	44.1	3.5	−3.2
	(31.1)	(6.1)	(3.5)	(3.0)	(1.1)	(5.1)
Async Effect	861.0	57.9	42.8	55.1	12.9	—
	(61.6)	(2.7)	(1.8)	(3.0)	(0.2)	
GitHub Agent	1004.7	71.1	49.6	62.6	17.0	—
	(95.6)	(4.2)	(2.1)	(2.7)	(0.4)	
OpenAI Codex	751.1	49.2	40.8	54.9	10.3	—
	(80.6)	(3.5)	(2.7)	(6.2)	(0.3)	
			Panel B. 0.1% Winsorization			
Autocomplete	2360.9	94.9	51.5	19.5	14.2	6.4
	(392.6)	(15.1)	(6.2)	(3.7)	(1.9)	(6.0)
Sync Effect	14,138.9	678.3	289.4	154.2	61.0	28.2
	(751.8)	(21.1)	(7.1)	(4.8)	(1.2)	(3.8)
Claude Code	13,574.3	911.7	540.6	296.7	106.3	55.1
	(1416.4)	(53.6)	(22.1)	(16.3)	(3.2)	(9.5)
GitHub Sync	12,120.7	587.6	214.5	115.2	59.2	22.5
	(811.9)	(25.0)	(7.8)	(5.1)	(1.7)	(5.0)
OpenAI Codex	16,283.4	664.7	260.0	130.1	41.2	14.1
	(1561.5)	(39.3)	(12.3)	(8.7)	(2.0)	(7.1)
Async Effect	5374.4	127.4	76.4	79.7	19.8	—
	(622.4)	(8.3)	(4.3)	(5.2)	(0.4)	
GitHub Agent	6071.0	146.2	80.6	87.4	23.3	—
	(902.0)	(12.0)	(5.1)	(5.9)	(0.5)	
OpenAI Codex	4624.2	112.0	77.4	80.6	17.5	—
	(876.7)	(11.6)	(6.8)	(9.7)	(0.5)	

Notes: This table reports event-study coefficients (weeks 21–30 bin, normalized by pre-period mean) measuring the effect of adopting AI coding tools on outcomes spanning the production hierarchy, re-estimating every outcome of Table 6 under one alternative specification per panel: Panel A removes the active-span restriction (Appendix D.3), and Panel B winsorizes at the 0.1% level instead of 1% (Appendix E.2). Standard errors, clustered at the developer level (developer-by-tool for the pooled rows), in parentheses. Outcome definitions and the async releases convention follow Table 6.

Table OA-8: Attenuation of Productivity Effects: Common Sample across Outcomes

	Outcomes (Weeks 21–30, %)					
	Lines Chg	Files Touched	Commits	PRs	Repos Touched	Releases
	Panel A. Intersection of All Six Outcome Samples					
Autocomplete	69.7	18.8	14.7	11.1	5.9	7.9
	(32.2)	(7.0)	(4.5)	(4.7)	(2.8)	(5.2)
Sync Effect	372.6	144.8	103.0	112.4	38.4	35.0
	(76.9)	(17.3)	(11.4)	(10.7)	(3.9)	(6.1)
Claude Code	479.3	178.2	135.1	151.1	48.6	42.4
	(88.9)	(20.6)	(13.7)	(14.3)	(4.9)	(7.5)
GitHub Sync	619.9	173.5	135.5	71.2	41.6	20.0
	(29.5)	(6.4)	(3.5)	(2.6)	(1.4)	(10.3)
OpenAI Codex	589.2	121.1	89.6	64.7	22.9	10.7
	(144.4)	(23.0)	(14.6)	(10.6)	(4.6)	(6.5)
Async Effect	534.7	46.1	36.3	68.4	14.2	—
	(74.6)	(3.9)	(2.3)	(3.4)	(0.4)	
GitHub Agent	645.0	54.2	42.9	73.3	17.9	—
	(111.8)	(4.9)	(2.7)	(3.2)	(0.5)	
OpenAI Codex	373.3	35.1	27.9	72.1	9.4	—
	(84.6)	(6.5)	(4.3)	(7.0)	(0.5)	
	Panel B. Intersection Excluding Releases					
Autocomplete	151.9	37.9	25.3	18.1	10.9	7.9
	(16.2)	(3.9)	(2.5)	(2.2)	(1.4)	(5.2)
Sync Effect	678.7	233.6	165.3	131.5	56.6	35.0
	(41.7)	(9.9)	(6.4)	(4.9)	(2.1)	(6.1)
Claude Code	890.7	307.1	226.2	179.2	74.6	42.4
	(55.0)	(13.1)	(8.6)	(6.6)	(2.7)	(7.5)
GitHub Sync	568.2	160.2	127.8	65.5	38.6	20.0
	(26.7)	(5.8)	(3.3)	(2.4)	(1.3)	(10.3)
OpenAI Codex	569.8	166.8	103.2	61.8	29.8	10.7
	(54.3)	(10.4)	(5.9)	(3.8)	(1.9)	(6.5)
Async Effect	534.7	46.1	36.3	68.4	14.2	—
	(74.6)	(3.9)	(2.3)	(3.4)	(0.4)	
GitHub Agent	645.0	54.2	42.9	73.3	17.9	—
	(111.8)	(4.9)	(2.7)	(3.2)	(0.5)	
OpenAI Codex	373.3	35.1	27.9	72.1	9.4	—
	(84.6)	(6.5)	(4.3)	(7.0)	(0.5)	

Notes: This table reports event-study coefficients (weeks 21–30 bin, normalized by pre-period mean, 1% winsorized) measuring the effect of adopting AI coding tools on outcomes spanning the production hierarchy, re-estimating every outcome of Table 6 on a balanced sample (Appendix E.2): Panel A restricts to matched pairs present in all six outcome samples, and Panel B to pairs present in the five samples excluding releases. Standard errors, clustered at the developer level (developer-by-tool for the pooled rows), in parentheses. Outcome definitions and the async releases convention follow Table 6.

Table OA-9: Productivity Effect by Outcome-Specific Decile — Pooled Sync Effect

	Post (Weeks 21–30) Treatment Effect, % of Pre-Period Treated Mean												
	Decile of Pre-Period Outcome										Summary		
Outcome	D1	D2	D3	D4	D5	D6	D7	D8	D9	D10	Raw D1–D10	Weight. D1–D10	Pooled Norm.
Lines Changed	11,854.8	1,642.9	744.3	324.7	217.4	136.5	102.5	54.3	28.0	1.7	1,510.7	1,305.6	957.5
	(1,110.3)	(145.7)	(52.1)	(24.4)	(13.8)	(8.4)	(5.4)	(3.0)	(1.5)	(0.2)	(113.1)	(95.5)	(23.2)
Files Touched	2,648.3	620.4	309.4	185.0	124.8	90.4	65.2	53.2	40.3	24.4	416.1	359.9	265.0
	(148.5)	(35.8)	(16.7)	(9.0)	(5.9)	(4.0)	(2.9)	(2.2)	(1.5)	(0.9)	(15.7)	(13.0)	(4.7)
Commits	1,272.7	531.3	299.6	194.4	128.1	93.5	68.0	53.5	39.8	23.6	270.4	232.4	153.2
	(60.2)	(30.7)	(14.3)	(9.2)	(6.0)	(4.3)	(2.9)	(2.2)	(1.5)	(0.8)	(7.2)	(5.8)	(2.7)
PRs Created	330.5	239.5	152.7	109.5	99.7	83.3	68.1	54.4	43.6	21.2	120.2	113.8	86.0
	(22.4)	(14.4)	(8.3)	(7.5)	(6.3)	(4.5)	(4.0)	(3.0)	(2.5)	(1.3)	(3.3)	(3.0)	(2.1)
Repos Touched	188.6	132.9	91.7	72.8	54.3	39.0	35.4	27.6	22.7	13.3	67.8	60.7	47.3
	(9.2)	(6.2)	(4.2)	(3.1)	(2.4)	(1.9)	(1.5)	(1.2)	(1.0)	(0.6)	(1.4)	(1.1)	(0.9)
Releases	54.8	61.1	31.2	41.0	18.6	22.9	17.8	14.1	14.4	8.2	28.4	27.8	19.8
	(14.5)	(13.2)	(10.1)	(7.9)	(7.2)	(6.4)	(5.8)	(4.9)	(4.1)	(2.7)	(3.1)	(3.0)	(2.9)

Notes: This table reports weeks 21–30 treatment effects for the pooled sync effect across the production hierarchy, estimated in levels separately within each decile of pre-period activity (Appendix E.2) and normalized by the decile-specific mean of the outcome's treated pre-period value, expressed in percent. Matched pairs are assigned to deciles using the pre-period mean of the row's outcome averaged across treated and control, avoiding regression to the mean. Treated–control differences are winsorized at the 1st and 99th percentiles before estimation. *Raw D1–10*: unweighted average of the decile effects. *Weighted D1–10*: deciles weighted by their share of users with a non-missing outcome at week 20, down-weighting high-attrition (typically low-activity) deciles like our main event studies. *Pooled Normalized*: the normalized event-study coefficient of the main specification (cf. Table 6), shown for comparison. Outcome definitions follow Table 6.

Table OA-10: Placebo: Productivity Effects Across Production Layers for Other Coding Tools

	Outcomes (Weeks 21–30, %)					
	Lines Chg	Files Touched	Commits	PRs	Repos Touched	Releases
GitHub Pro	−7.7	−25.6	−7.3	23.4	1.1	12.1
	(84.0)	(17.6)	(10.7)	(7.9)	(5.9)	(22.1)
Docker	−82.5	8.4	6.6	13.6	3.8	−5.3
	(13.0)	(2.4)	(1.4)	(1.2)	(0.6)	(2.6)

Notes: This table reports event-study coefficients (weeks 21–30 bin, normalized by pre-period mean, 1% winsorized) measuring the effect of adopting the two non-AI placebo tools, re-estimating every outcome of Table 6 for GitHub Pro and Docker (Appendix E.3). Standard errors, clustered at the developer level, in parentheses. Outcome definitions follow Table 6.

Table OA-11: Productivity Effects by Matching Specification

	Paper spec		Contemp.		Acct-year		Unif. commit		Unif. PR		Unif. both	
	Commits	PRs	Commits	PRs	Commits	PRs	Commits	PRs	Commits	PRs	Commits	PRs
Autocomplete	30.2	18.3	26.5	11.1	28.6	24.5	15.9	21.4	31.9	18.1	30.2	18.3
	(2.7)	(2.2)	(2.7)	(2.2)	(2.7)	(2.1)	(1.9)	(2.0)	(2.7)	(2.1)	(2.7)	(2.2)
Sync Effect	153.2	86.0	85.5	63.6	151.0	85.8	137.9	89.0	140.4	94.8	138.5	89.5
	(2.7)	(2.1)	(2.7)	(2.2)	(2.7)	(2.1)	(2.7)	(2.4)	(3.2)	(2.5)	(3.2)	(2.4)
Claude Code	295.5	178.9	225.8	138.6	303.4	172.1	279.2	174.3	269.5	186.3	268.6	185.6
	(7.7)	(6.6)	(7.5)	(6.4)	(7.7)	(6.5)	(7.9)	(7.2)	(10.3)	(8.3)	(10.5)	(8.3)
GitHub Sync	126.3	65.5	59.7	45.5	122.2	64.9	114.4	66.5	113.8	78.7	114.8	72.5
	(3.2)	(2.3)	(3.5)	(2.6)	(3.3)	(2.4)	(3.2)	(2.7)	(3.8)	(2.9)	(3.9)	(2.9)
OpenAI Codex	118.8	61.2	50.6	45.2	108.8	60.0	95.8	71.9	129.9	74.3	126.3	67.4
	(4.6)	(3.8)	(4.7)	(4.0)	(4.7)	(3.8)	(5.1)	(4.7)	(4.9)	(4.2)	(4.8)	(4.1)
Async Effect	60.7	68.5	56.0	69.4	59.4	65.9	56.4	68.0	58.7	64.0	59.3	60.8
	(2.4)	(3.4)	(2.2)	(4.0)	(2.4)	(3.4)	(2.4)	(4.8)	(3.0)	(4.8)	(2.9)	(4.6)
GitHub Agent	62.4	73.2	59.5	74.5	63.4	73.9	56.7	74.4	58.3	59.3	60.9	59.9
	(2.5)	(3.2)	(2.6)	(3.6)	(2.7)	(3.2)	(2.6)	(4.5)	(3.5)	(3.5)	(3.7)	(3.6)
OpenAI Codex	63.3	72.4	57.1	73.6	59.8	64.0	61.0	71.1	61.4	78.7	61.3	72.1
	(4.0)	(7.0)	(3.5)	(8.3)	(3.8)	(7.0)	(3.9)	(9.9)	(4.1)	(8.7)	(3.9)	(8.3)

Notes: This table reports weeks 21–30 treatment effects on commits and pull requests, normalized by each pair's pre-period mean and 1% winsorized, under the alternative matching specifications of Appendix E.1: the paper baseline, contemporaneous controls, matching additionally on account-creation year, and uniform adoption-week activity requirements (a commit only, a pull request only, or both, imposed on every adopter and control). Standard errors, clustered at the developer level (developer-by-tool for the pooled rows), in parentheses. Only commits and pull requests are reported because per-commit and repository metadata are collected only for baseline matched controls (Appendix C.5), so the remaining outcomes are not estimable for re-matched controls.

Table OA-12: Robustness to Pre-period Window

	−22		−32		−42		−52	
	Commits	PRs	Commits	PRs	Commits	PRs	Commits	PRs
Autocomplete	27.8	20.1	20.9	24.8	22.3	24.4	24.4	31.9
	(2.9)	(3.0)	(2.7)	(3.3)	(2.7)	(3.3)	(2.6)	(3.6)
Sync Effect	128.8	89.9	137.8	117.1	126.0	125.5	138.1	142.5
	(3.3)	(2.9)	(3.0)	(3.1)	(2.9)	(3.3)	(2.9)	(3.5)
Claude Code	266.7	208.1	260.2	238.7	250.2	226.5	245.3	254.9
	(8.9)	(9.1)	(8.3)	(9.7)	(7.6)	(8.7)	(7.2)	(10.0)
GitHub Sync	96.0	59.1	95.5	91.1	91.5	95.3	98.4	100.3
	(3.6)	(3.1)	(3.4)	(3.7)	(3.2)	(3.8)	(3.1)	(3.8)
OpenAI Codex	100.1	63.9	125.1	77.6	100.3	104.2	127.0	134.4
	(6.0)	(5.4)	(5.5)	(5.2)	(5.4)	(6.3)	(5.6)	(6.9)
Async Effect	63.4	126.2	76.4	167.9	73.6	204.1	69.5	244.0
	(3.4)	(9.6)	(4.8)	(15.1)	(4.9)	(17.2)	(4.1)	(22.0)
GitHub Agent	61.2	112.2	67.2	130.5	61.7	169.5	62.9	189.6
	(3.8)	(6.3)	(4.2)	(6.7)	(3.4)	(9.4)	(4.2)	(12.0)
OpenAI Codex	69.4	162.4	89.3	241.5	90.1	274.2	80.6	343.9
	(5.5)	(21.2)	(8.4)	(34.2)	(9.0)	(37.7)	(7.0)	(48.5)

Notes: This table reports weeks 21–30 treatment effects on commits and pull requests, normalized by each pair's pre-period mean and 1% winsorized, re-matching and re-normalizing with longer pre-period windows (Appendix E.1): each column pair matches on total commits over a window beginning −22, −32, −42, or −52 weeks before adoption; the normalization window moves with the matching window, as does the requirement of at least one week with positive commits at or before the window's start. Standard errors, clustered at the developer level (developer-by-tool for the pooled rows), in parentheses. Only commits and pull requests are reported because per-commit and repository metadata are collected only for baseline matched controls (Appendix C.5), so the remaining outcomes are not estimable for re-matched controls.

References for Online Appendix

Goodman-Bacon, A. (2021). Difference-in-Differences with Variation in Treatment Timing. *Journal of Econometrics 225*(2), 254–277.

Jones, C. I. (2026). AI and Our Economic Future. *Journal of Economic Perspectives 40*(3), 3–22.

Milgrom, P. and J. Roberts (1996). The LeChatelier Principle. *American Economic Review 86*(1), 173–179.

Samuelson, P. A. (1947). *Foundations of Economic Analysis*. Harvard University Press.

Sun, L. and S. Abraham (2021). Estimating Dynamic Treatment Effects in Event Studies with Heterogeneous Treatment Effects. *Journal of Econometrics 225*(2), 175–199.