NBER WORKING PAPER SERIES

WRITING CODE VS. SHIPPING CODE:
PRODUCTIVITY EFFECTS ACROSS GENERATIONS OF AI CODING TOOLS

Mert Demirer
Leon Musolff
Liyuan Yang

Working Paper 35275
http://www.nber.org/papers/w35275

NATIONAL BUREAU OF ECONOMIC RESEARCH
1050 Massachusetts Avenue
Cambridge, MA 02138
May 2026

We thank Joel Becker, Alexander Bick, Adam Blandin, David P. Byrne, Tom Cunningham, Jan De Loecker, Anders Humlum, Sanjog Misra, Frank Nagle, Sida Peng, Nate Rush, Suproteem Sarkar, Max Schnidman, and Chad Syverson. 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, and the Columbia MAD conference. This project was funded in part by the Mack Institute For Innovation Management at Wharton and the Chicago AI Incubator. The views expressed herein are those of the authors and do not necessarily reflect the views of the National Bureau of Economic Research.

At least one co-author has disclosed additional relationships of potential relevance for this research. Further information is available online at http://www.nber.org/papers/w35275

Writing Code vs. Shipping Code: Productivity Effects Across Generations of AI Coding Tools
Mert Demirer, Leon Musolff, and Liyuan Yang
NBER Working Paper No. 35275
May 2026
JEL No. D24, L86, O33

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 100,000 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 40%, 140%, and 180%. These gains, however, attenuate sharply across the production hierarchy: the 180% cumulative effect falls to 50% 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 0.25 between AI and human effort, which indicates strong complementarities. We further confirm these results across four major app marketplaces, finding a moderate 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.

Mert Demirer
Massachusetts Institute of Technology
Department of Economics
and NBER
mdemirer@mit.edu

Liyuan Yang
Massachusetts Institute of Technology
lyyang@mit.edu

Leon Musolff
University of Pennsylvania
Wharton School
Business Economics,
and Public Policy
and NBER
lmusolff@wharton.upenn.edu

1 Introduction

A distinguishing feature of generative AI, relative to earlier technologies, is the speed of technological progress and diffusion (Bick et al., 2024; Humlum and Vestergaard, 2025). As a result, model capabilities have improved rapidly, and firms have deployed AI across a growing range of applications (OpenAI, 2025). 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. First, how do productivity effects evolve across successive generations of AI tools? Second, to what extent do productivity gains at the task level translate into final output?

We study these questions in the context of 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 (Eloundou et al., 2024; Tomlinson et al., 2025). Software development provides an ideal setting for two reasons. First, it has experienced multiple waves of widely adopted AI-based tools. These include an *autocomplete* function that suggests code as the developer types, *sync agents* that write and edit code alongside the developer in real time, and, more recently, *async agents* that can be assigned a task and work autonomously without developer oversight. Second, its production process involves a well-defined sequence of stages—writing code, integrating changes across files, reviewing and merging pull requests, and managing releases—that allows us to measure productivity at different points along the production chain.

Our paper makes two contributions to understanding the economic impact of AI. First, we contribute to the literature on how AI affects labor productivity in specific tasks (Noy and Zhang, 2023; Brynjolfsson et al., 2025; Cui et al., 2026; Dell'Acqua et al., 2026) by providing evidence from more than 100,000 software developers on GitHub across multiple AI tools. Our analysis spans multiple generations of these tools from 2022 to 2026, allowing us to track how the effects evolve as model capabilities advance, rather than capturing a single snapshot of the technology.

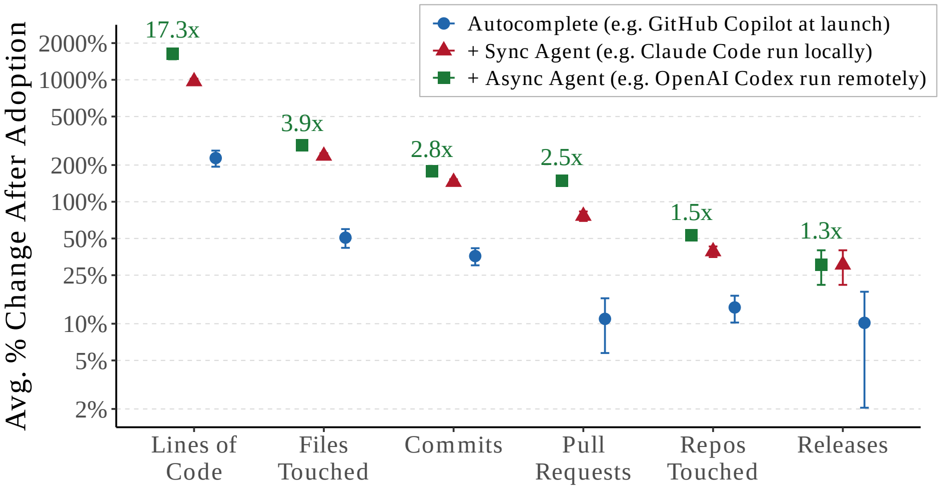
In our second contribution, we study the extent to which these *task-level productivity* gains translate into *final output*. While most estimates of task-level productivity effects of AI are quite large (15–50%, see the review by Fruits and Stout, 2026), several papers suggest that these gains do not necessarily translate into proportional increases in aggregate output (Acemoglu, 2025; Jones, 2026). A prominent argument is the *bottleneck hypothesis*, or *weak links*: AI may be highly productive at specific tasks, but overall output is limited by complementary tasks that are still performed by humans whose productivity is unchanged (Kremer, 1993; Jones, 2011, 2026). To study this phenomenon, we extend the weak-link framework to a hierarchical model in which software output is produced through the sequential aggregation of tasks within a single production process, and provide one of the first empirical tests of this hypothesis. We find that task-level productivity effects attenuate substantially as we move from more granular to higher-level outcomes, suggesting that AI's large productivity effects at earlier stages (writing code) are attenuated by human bottlenecks at later stages (reviewing pull requests and launching apps).

In our model of software production, output from each layer is combined with the input from the subsequent layer through a CES production function. AI tools can affect each layer of the production process in one of two ways: by augmenting it (making each unit of human effort more productive at that layer) or by partially automating it (rendering direct human effort at that layer unnecessary but still requiring human review). For instance, *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 this model is the elasticity of substitution between upstream output generated by AI and subsequent human effort: the less substitutable the two are, the more AI's productivity gains are bottlenecked by the human stages downstream, and the more they attenuate as they propagate along the production chain. In the case of complementarities, even unbounded automation of the upstream layer yields only finite productivity gains, similar to the prediction of the weak-links literature (Kremer, 1993; Jones, 2011).

To take this framework to the data, we combine public GitHub data with internal Microsoft telemetry to construct activity measures that correspond to each layer of the production hierarchy—from lines of code through releases—and to identify when developers first adopt each tool. Combining publicly observable AI usage on GitHub with internal subscription data lets us track multiple tools across different 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 control user in the same calendar week one year earlier, avoiding contamination from widespread private adoption of AI tools among contemporaneous non-adopters.

We use several strategies to address the endogeneity concerns inherent in observational data. 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 12 weeks. Second, we validate our results with placebo tests that apply the same event-study design to non-AI coding tools. Third, we document the absence of differential pre-trends between treated and control users across all outcomes, supporting the parallel trends assumption. Fourth, we compare our autocomplete estimates with the prior field-experimental evidence

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, distinct repositories, and releases. The underlying estimates are from Table 5. "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.

of Cui et al. (2026), who estimate the effect of the same tool in the same time period. Fifth, we examine the dynamics of the treatment effects and find that they track the timing of major model launches. Sixth, to address the concern that we rely only on public-repository data, we validate some of our results using limited data on AI usage in private repositories.

Our results are summarized in Figure 1. We find that AI coding tools substantially increase developer task-level productivity, with effects that grow across successive generations of tools. Focusing on the number of commits, autocomplete increases task-level productivity by approximately 40%; the cumulative effect including sync agents rises to approximately 140%; and additional use of async agents raises it to 180% through agent-authored commits. These effects are larger for less active developers, though they remain substantial throughout the activity distribution. Taken together, these results point to a substantial productivity shift in software development with the rise of agentic AI tools.

How do these task-level productivity increases translate into final output? Our estimates across the layers of the production hierarchy reveal a clear pattern of attenuation. After adopting AI tools, developers write substantially more code and work on more features, but this translates into much smaller increases in shipped software. Sync agents lead to a 741% increase in lines of code and a 65% increase in pull requests, yet releases rise by only 20%. Autocomplete shows the same pattern: a 228% effect on lines of code attenuates to 36% on commits and just 10% on releases. We also confirm these developer-level estimates with total GitHub activity: new repositories and pull requests have increased substantially since early 2025, in a pattern consistent with the attenuation we estimate.

This pattern of attenuation across the production hierarchy is consistent with the prediction of our production model. To quantify potential complementarities and the mechanism behind attenuation, we next map our estimates to the model's parameters. The estimated attenuation pattern is consistent with an elasticity of substitution of 0.25 between upstream output and human effort at each layer, 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.

While GitHub coding activity is useful for understanding how AI affects software production, it is still a measure of developer-side output: it does not directly tell us whether new apps reach consumers or how much they are actually used. To address these questions, we assemble monthly panels for four of the largest application 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 application appeared and a measure of cumulative end-user engagement (number of ratings or downloads). With these data, we quantify whether the number of new applications has increased over time and, if so, whether this translates into greater usage.

Across marketplaces, we see a broad increase in new applications since mid-2025, though the magnitude varies (a sharp acceleration on the Apple App Store and the Chrome Web Store; a milder increase on the Google Play Store; and little change on SourceForge). Despite this expansion in supply, we find that total app usage within the first three months of launch has not increased in any of the four marketplaces. 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 strongly support this hypothesis either: the share of new applications that fail to reach even a modest audience has risen across marketplaces, suggesting that the supply-side expansion is concentrated in applications with little to no user base. This pattern admits two interpretations: either the marginal applications are of low quality, or there is an additional bottleneck on the consumer side—discovery and adoption of new applications take time—that prevents supply-side gains from translating into usage. Our data currently cannot distinguish between the two.

Overall, our results have important implications both for the current impact of AI and for the mechanisms that will shape the economy going forward. We document that the large task-level productivity effects of AI translate into a much smaller impact on final output because of bottlenecks in the production function. These gains face further compression at the end-user stage, where applications must reach and be used by consumers. How much of this constraint loosens going forward will depend both on future model capabilities—whether AI tools can produce higher-quality code that requires less review or substitute further down the hierarchy—and on the diffusion of these tools across stages of production.

It is worth mentioning some limitations of our study. First, our ability to measure software quality is limited to indirect inference from consumption data such as ratings and downloads. Second, while our combination of GitHub activity and four major application marketplaces captures a large segment of the software industry, it does not cover some other important parts of the market—most notably enterprise and internal-only software. Third, although we can empirically test some general principles about the impact of AI using our setting, software is by far the most advanced application of generative AI to date, and our quantitative estimates may not extrapolate to other domains.

The remainder of the paper is organized as follows. Section 2 reviews related literature. Section 3 provides background on developer workflows and AI coding tools. Section 4 develops the production model. Section 5 describes the data and sample construction. Sections 6 and 7 present the productivity estimates across tool generations and across the production hierarchy. Section 8 examines aggregate software output. Section 9 concludes.

2 Literature

Our work relates to four strands of literature: AI and productivity, task-based automation and multi-stage production, the debate over whether task-level productivity gains translate into aggregate productivity, and the effects of AI on entry in digital marketplaces.

A growing literature studies the effects of generative AI tools on productivity across settings, including coding (Peng et al., 2023; Hoffmann et al., 2024; Chen et al., 2025; Cui et al., 2026; Sarkar, 2026; Becker et al., 2025; He et al., 2026), customer support (Brynjolfsson et al., 2025), law (Choi and Schwarcz, 2023), consulting (Dell'Acqua et al., 2026), writing (Noy and Zhang, 2023), entrepreneurship (Otis et al., 2025), and general knowledge work (Kreitmeir and Raschky, 2024; Dillon et al., 2025; Jabarian and Henkel, 2026). With some exceptions, these papers find productivity effects on the order of 15–50% (Fruits and Stout, 2026). However, generative AI has advanced rapidly, and most of these estimates pertain to early-generation tools—primarily autocomplete and chatbot interfaces. Systematic evidence on more recent agentic tools remains limited and mixed: Becker et al. (2025) find that agentic coding tools reduce experienced developers' productivity by 19% in an RCT, while other studies report productivity gains of 35–40% (Sarkar, 2026; Chen et al., 2025). We contribute to this literature by comparing productivity effects across three successive generations of tools using a consistent methodology and large-scale observational data.

Our conceptual framework builds on the task-based approach to automation (Acemoglu and Restrepo, 2019), 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), and Garicano et al. (2026) consider how automation propagates through interdependencies and chains of tasks.[1] 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). Aghion et al. (2019) and Jones (2026) formalize this for AI specifically, showing that if automated and non-automated tasks are complements, even full automation of a subset of tasks yields finite output gains. We apply this logic vertically within a single production process: the "tasks" are the layers of the software production hierarchy (code, commits, pull requests, releases), and the weak link is the human-bottlenecked layer with the least capacity. This yields a specific, testable prediction—that productivity gains attenuate through the hierarchy—which we confirm empirically.[2]

Our aggregate analysis connects to a longstanding debate about whether productivity gains at the micro level translate into aggregate output growth (Brynjolfsson et al., 2021). Solow (1987) famously noted the disconnect between rapid IT investment and sluggish productivity statistics. We examine not only developer-level productivity but also platform-wide coding activity and new application creation across software marketplaces. This allows us to provide early evidence on whether the large task-level effects documented in this paper and others are beginning to manifest in aggregate software output.

Finally, a growing literature studies how generative AI reshapes output and consumption on digital platforms. Reimers and Waldfogel (2026) find that Large Language Models (LLMs) tripled new book releases on Amazon between 2022 and 2025 while average quality declined; Goldberg and Lam (2026) document a large supply expansion along-

---

[1] Other related contributions include Autor and Thompson (2025), who model how automation reshapes occupations via task bundling and required expertise; Freund and Mann (2026), who develop a general-equilibrium model of job transformation in which workers sort by comparative advantage; Restrepo (2025) extends the weak-link/bottleneck logic to AI by distinguishing bottleneck and accessory work in an AGI economy; and Korinek and Suh (2024) study how a race between automation and capital accumulation shapes wages and output as AI approaches AGI, with fixed factors potentially generating growth bottlenecks.

[2] Our model is also related to hierarchical models of knowledge work, in particular Garicano (2000) and Ide and Talamàs (2025). While their hierarchies are in skill—less knowledgeable workers handle routine tasks while more knowledgeable workers handle exceptions—we consider a hierarchy in the production process, in which more granular outputs are aggregated into higher-level outputs.

side crowd-out of incumbents on a stock-images marketplace; and Zhou and Lee (2024) find similar entry and engagement effects on an image-sharing platform. We extend this work to software, documenting an analogous supply expansion across major application marketplaces—with consumption that has not kept pace.

3 Background

In this section, we briefly outline a typical developer workflow and provide a short history of generative AI tools designed to assist software developers, which we study in detail in subsequent sections.

3.1 Developer Workflow

Software developers write computer code using integrated development environments (IDEs), such as Visual Studio Code (VS Code) and JetBrains IDEs (e.g., IntelliJ or PyCharm). IDEs provide features—such as syntax highlighting, error checking, and debugging tools—that facilitate writing code. Code is written and organized in plain-text source *files* that developers structure into projects and modules.

Software development projects are typically managed using version-control systems, most commonly Git. In a version-tracking system, developers periodically bundle a set of semantically related code changes, provide a brief description, and record the result as a *commit*. Commits are the fundamental unit of version control: each commit represents a reversible change to the codebase.

Complex software projects typically involve multiple developers working simultaneously. Tasks are commonly 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 there is agreement on what needs to be implemented, a developer is assigned the issue and implements the required changes.

To facilitate parallel work, version-tracking systems allow developers to create separate *branches* of the codebase. Each developer can work independently on a branch without immediately incorporating changes made by others. Once a developer has completed work on an issue, they submit a *pull request*, which signals that the proposed changes are ready for review. Other developers can then inspect the code, request modifications, and ultimately merge the changes into the project's main branch. Each project lives in its own *repository*, and a single developer may contribute to several repositories at once. Finally, each repository can be packaged into *releases*—compiled code associated with a version number and a changelog.

3.2 A Brief History of Generative AI Tools in Software Development

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

Autocomplete Even before the release of ChatGPT in November 2022, GitHub launched GitHub Copilot in June 2022, the first widely-adopted LLM-based coding assistant tool.[3] The initial version of Copilot integrated directly with developers' IDEs and served as an intelligent autocomplete tool. As developers write code or plain-text comments, Copilot analyzes the context and generates relevant code snippets, comments, and documentation. Prior field experiments suggest that autocompletion tools increase developer productivity by roughly 26% (Cui et al., 2026).[4]

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. 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. Throughout this process, the developer monitors the agent's behavior, reviews intermediate outputs, approves proposed changes, and retains responsibility for integrating those changes into the codebase. Because the developer supervises the agent's work in real time, these agents are called sync agents.

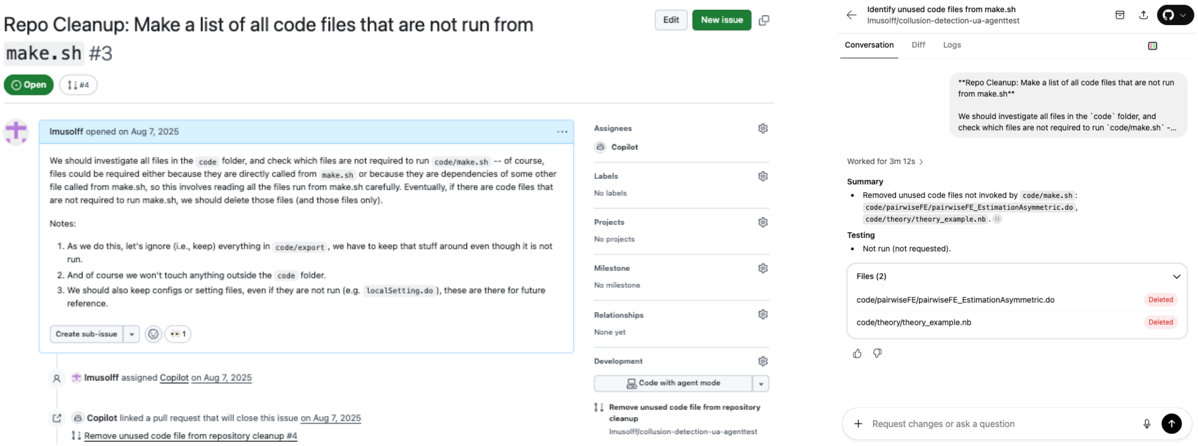
Sync coding agents also differ in how they are accessed and embedded in developers' workflows. Some are integrated directly into IDEs—such as Cursor or GitHub Copilot—where interaction occurs through an in-editor interface and the agent has immediate access to the active files and project context. Others are invoked through a command-line interface or a dedicated external interface—such as Claude Code or OpenAI Codex—where developers submit prompts and agents edit the files outside a dedicated IDE.

---

[3] While GitHub Copilot was the first to launch AI-assisted coding features integrated directly into IDEs, many others have since followed with similar offerings, such as Codeium (late 2022), Cursor (March 2023), and Amazon CodeWhisperer (April 2023).

[4] Following the launch of ChatGPT in November 2022, developers also quickly adopted AI chatbots for coding-related tasks such as explaining unfamiliar code, debugging errors, and generating snippets. 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 in this paper.

Figure 2: Assigning Tasks to Async Coding Agents



Repo Cleanup: Make a list of all code files that are not run from `make.sh` #3

Imusolff opened on Aug 7, 2025

We should investigate all files in the `code` folder, and check which files are not required to run `code/make.sh` -- of course, files could be required either because they are directly called from `make.sh` or because they are dependencies of some other file called from make.sh, so this involves reading all the files run from make.sh carefully. Eventually, if there are code files that are not required to run make.sh, we should delete those files (and those files only).

Notes:

1. As we do this, let's ignore (i.e., keep) everything in `code/export`, we have to keep that stuff around even though it is not run.
2. And of course we won't touch anything outside the `code` folder.
3. We should also keep configs or setting files, even if they are not run (e.g. `localSetting.do`), these are there for future reference.

Imusolff assigned Copilot on Aug 7, 2025

Copilot linked a pull request that will close this issue on Aug 7, 2025

Remove unused code file from repository cleanup #4

Remove unused code file from repository cleanup
Imusolff/collusion-detection-ua-agenttest

(a) Issue Assigned to Copilot

Identify unused code files from make.sh
Imusolff/collusion-detection-ua-agenttest

Repo Cleanup: Make a list of all code files that are not run from make.sh

We should investigate all files in the \`code\` folder, and check which files are not required to run \`code/make.sh\` -...

Worked for 3m 12s

Summary

- Removed unused code files not invoked by `code/make.sh`: `code/pairwiseFE/pairwiseFE_EstimationAsymmetric.do`, `code/theory/theory_example.nb`.

Testing

- Not run (not requested).

Files (2)
code/pairwiseFE/pairwiseFE_EstimationAsymmetric.do
code/theory/theory_example.nb

Request changes or ask a question

(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 GitHub Copilot Coding agent directly on GitHub.com; on the right-hand side, a developer has assigned a task to OpenAI Codex on OpenAI's website.

Async Agent A more recent development is the emergence of async coding agents. Unlike sync agents, which operate interactively in the IDE and require frequent developer input, async agents are designed to operate autonomously for extended periods with little or no ongoing human interaction. Developers typically assign these agents a high-level task—such as implementing a feature, fixing a bug, or refactoring part of a codebase—and the agent independently plans and executes the required steps. As a result, async agents are less iterative in nature and can take on substantially larger and more complex tasks than sync agents.[5]

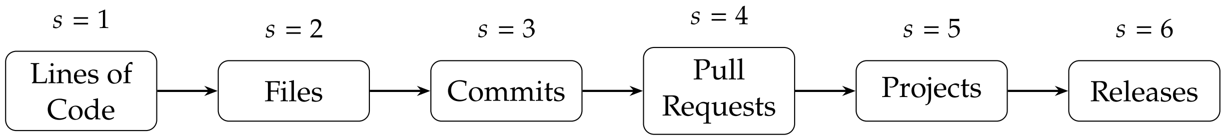
Async coding agents have rapidly gained prominence. On May 19, 2025, GitHub introduced the *GitHub Async Agent* (hereafter, the Copilot async agent), which operates directly within GitHub. The agent is invoked by assigning it an issue, in the same way as assigning a task to a human collaborator; see Figure 2(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; see Appendix Figure OA-1 for an example.

Other tools, such as OpenAI Codex and Claude Code, also support async agent func-

---

[5] In practice, the boundary between sync and async usage is increasingly blurred. Tools such as Codex and Claude Code can operate in either mode depending on the task: a developer may interact iteratively on small subtasks (sync) or assign a larger task that the agent executes autonomously and returns as a pull request (async). The practical distinction thus depends more on the complexity of the assigned task and degree of developer supervision than on a strict technological boundary.

Figure 3: The Software Production Hierarchy



Notes: This figure depicts the software production hierarchy used throughout the paper. Each layer s aggregates the output of the layer below: lines of code combine into files, files into commits, commits into pull requests, pull requests into projects, and projects into releases.

tionality by autonomously modifying code and producing pull requests.[6] Unlike GitHub Copilot's coding agent, however, task assignment does not occur through GitHub issues; instead, developers initiate tasks through an external interface (e.g., via OpenAI's web interface for Codex). Figure 2(b) shows an example of a developer assigning a task to Codex, and Appendix Figure OA-2 displays the resulting pull request.

4 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.

4.1 Setup

Production proceeds through S layers, ordered from most granular to most aggregate (Figure 3). At the lowest layer, the developer writes lines of code. At each subsequent layer $s \geq 2$, units of layer-$(s-1)$ output are aggregated into higher-level units of layer-s output. Assembling lines into a correct file, bundling files into a coherent commit, or integrating commits into a pull request each requires effort—for review, coordination, and design decisions. This effort may come from the developer, from an AI tool, or from a combination of both.

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

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

---

[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.

where e_1 denotes the effective input devoted to writing code, which may include both human effort and AI-generated output and is formally defined below. At each subsequent layer $s \geq 2$, production combines upstream input with effective input 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 the layer below, e_s is the effective input at layer s, $\alpha_s \in (0, 1)$ is the share parameter, and $\sigma_s > 0$ is the elasticity of substitution between upstream input and effective input. Final output is $Y \equiv y_S$.[7]

Effective input. At each layer s, the effective units of input combine the developer's own effort with AI-generated output:

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

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 output (h_s^r), and $a_s \geq 0$ is the quantity of AI-generated output prompted by the developer 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. The remaining parameters capture AI's productivity effects: $B_s \geq 1$ captures how much AI increases the output the developer produces per unit of effort, and $\phi_s \geq 0$ governs how much human input is needed to review the AI output. 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. With this normalization, B_s represents the factor by which AI augments developer productivity at layer s. We refer to e_s as "effective input" throughout, whether it originates from the human, the AI, or both.

The 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 generates output* a_s *at layer s—for example, autonomously writing code or creating pull requests—but this output requires human review before it is useful. The Leontief structure* $\min(a_s,\ \phi_s h_s^r)$ *implies that for each*

---

[7] In our empirical setting, the observable final output is a binary indicator of release, which is not standard in a production function. We instead interpret Y as the quality of the released software, which determines actual consumption.

unit of AI output, the developer must devote effort proportional to $1/\phi_s$ *to reviewing it. Automation is only partial because human input remains essential: without human review, AI does not produce any output.*

Definition 3 (Full automation)**.** *In the limit* $\phi_s \to \infty$*, AI output requires no human review and* $\min(a_s, \phi_s h_s^r) \to a_s$*. Automation is full because the human is no longer needed at layer* s*.*

4.2 Discussion

AI's productivity effects. The three mechanisms differ in the magnitude of the productivity gain they deliver. Under full automation, the gain is in principle infinite: output can be produced with no human input. Under augmentation, the gain is proportional to B_s, which depends on how much AI improves the developer's own effort—for example, the speed of writing code. Under partial automation, the gain depends on the AI quality parameter ϕ_s, which determines how much review effort each unit of AI output requires.

Augmentation vs. automation. In the model, augmentation and automation do not coexist within a layer. Because h_s^p and h_s^r enter e_s as perfect substitutes, 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 output (a_s), but not to a mix of the two. This is by design: contemporary AI coding tools are typically used in one mode at a time, not blended within a layer. The distinction between h_s^p and h_s^r is therefore largely semantic—the same productivity gain at layer s can in principle be delivered through either channel—but we keep both because they map to economically distinct use cases of AI, and because h_s^r makes explicit the review and integration effort that defines how developer work has changed.[8]

Interpretation. While stylized, the model captures several key features of software development. First, production is *hierarchical*: each layer aggregates many lower-level units into fewer higher-level ones, reflecting the natural structure of codebases. Second, AI can enter at any layer through two distinct channels, augmentation or automation, and different tools activate different channels at different layers. Third, unless there is full automation ($\phi_s \to \infty$), the human is never fully absent: even when AI generates output autonomously, the Leontief structure ensures unreviewed AI is unproductive.

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

---

[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: AI Coding Tools and the Production Hierarchy

	AI's role at each layer					
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 increases the developer's productivity ($B_s > 1$). "Partial automation" denotes layers where AI generates output subject to human review ($a_s > 0$, $e_s = \min(a_s, \phi_s h_s)$). "—" indicates no AI involvement.

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 with multiple nests (see Appendix A.1 for the derivation). Nested CES is a natural representation for hierarchical production because of its homotheticity: inputs within each nest aggregate into an intermediate input that enters higher-level nests as a single argument (Shephard, 1953; Demirer, 2025). 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.

4.3 Mapping AI Coding Tools to the Production Function

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 lines of code faster by proposing code snippets as they type. In the model, autocomplete operates exclusively through the augmentation channel at layer 1:

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

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 remains purely human ($\phi_s = 0$ so $a_s = 0$ for all s, and $B_s = 1$ for $s \geq 2$). The productivity gain in code generation must therefore pass through every subsequent layer before reaching final output.

Sync agents. Sync agents can autonomously generate and edit code across multiple 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:

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

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:

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

Relative to autocomplete, the sync agent operates through multiple channels: it generates output at layers 1–2 to be reviewed by the developer 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:

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

The key difference from sync agents is that $\phi_s > 0$ so $a_s > 0$ at more layers—the async agent produces commits (a_3) and pull requests (a_4) autonomously, not just code and files.

4.4 The Elasticity of Substitution and Shock Propagation

We now explore how the AI-generated productivity gains described in the previous section translate into final output. We take a partial-equilibrium approach: we ask what happens when AI generates productivity gains at upstream layers and human inputs in all downstream layers are fixed. While this is a significant simplification, since in practice developers will reoptimize their effort across layers in response to AI productivity gains, the fully optimal allocation is not tractable in closed form. This partial-equilibrium approach still captures the main channels and economic logic of the problem.

Consider an AI-driven productivity gain only at layer $s-1$ that raises output there by a factor $A > 1$. Holding all other inputs fixed, the pass-through to the next layer is (see Appendix A):

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

where $\tilde{y}_s(A)$ denotes the post-shock value of y_s. 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 developer effort at that layer. The second parameter, ρ_s, is the curvature parameter, where σ_s is the elasticity of substitution between layer-($s-1$) output and effective input at layer s. Four results follow.

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

(a) strictly increasing in θ_s, the output elasticity of y_s with respect to upstream input;

(b) strictly increasing in σ_s, the elasticity of substitution between upstream and effective input at layer s.

(See Appendix A for the proof.)

This proposition shows that the higher the output elasticity of upstream input θ_s, the higher the pass-through. For example, consider a layer where the upstream output dominates the production of layer output—such as aggregating lines of code into commits. Here θ_s is likely close to one, so an upstream productivity gain has a large pass-through to layer output.

Holding θ_s fixed, the key parameter determining the rate of propagation is the elasticity of substitution σ_s. When $\sigma_s > 1$ (substitutes), upstream input can mostly replace effective input, so abundant code compensates for limited review capacity, leading to slow attenuation of productivity gains across layers. When $\sigma_s < 1$ (complements), upstream input cannot easily substitute for effective input—an abundance of code is of limited value without commensurate review effort, and in the limit $\sigma_s \to 0$ the technology approaches Leontief, $y_s \to \min(y_{s-1}, e_s)$, where there is no productive gain from AI since the output is proportional to human input.

This proposition has 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)**.** *Suppose at every layer s, upstream input has positive weight. Then 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. The cumulative effect at layer s is the composition of pass-throughs at each intervening layer.*

The intuition is straightforward: at each layer, the upstream output must be combined with effective input whose quantity has not changed, so only a fraction of the gain passes through. The higher the layer, the more such bottlenecks the shock must traverse, and the more the gain is compressed.

Corollary 2 (Layer of entry)**.** *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.*

This result implies 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. In the context of Table 1: agents (entering at layer 4) face only two layers of attenuation before reaching releases, while autocomplete (entering at layer 1) faces five.

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.*** *Consider a shock propagating through multiple layers with common σ and θ, 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 layer. Define the per-layer log-survival ratio $R_s \equiv \ln G_{s+1} / \ln G_s \in (0, 1)$, the fraction of the log 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) ***Full automation.*** *When upstream input grows without bound ($A \to \infty$), the output gain converges to:*

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

(See Appendix A.4 for the proof.)

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 infinite productivity gains in an upstream layer (full automation) yield bounded gains if the downstream layers are not also fully automated. 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 (Jones, 2011; Kremer, 1993; 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.[9] 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), which holds under unit elasticities, to a nonlinear environment in which substitution elasticities shape how shocks propagate through the network.

5 Data and Summary Statistics

This section describes our data sources, the construction of our sample, how we identify adopters of each AI tool, and the resulting summary statistics.

5.1 Data Sources

Our analysis draws on three data sources: publicly available GitHub data on activity in public repositories, Microsoft-internal data on the adoption and usage of AI-based coding

---

[9] Jones (2026) shows that fully automating tasks with a spending share s raises output by a factor of at most $\frac{1}{1-s}$; our bound (10) generalizes this result to arbitrary $\sigma < 1$, and reduces to Jones's formula when $\sigma = 1/2$ (see Appendix A.5).

tools available in GitHub Copilot, and data on four major software marketplaces. Detailed data collection procedures are provided in Appendix B.

GitHub Data GitHub is the dominant platform for hosting and collaborating on software projects, with over 180 million developers and 395 million public repositories.[10] 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 (Hoffmann et al., 2024; Wright et al., 2023). We later use our limited data on closed-source AI activity to assess the external validity of our public-activity results.

GitHub records detailed developer activity data—including commits, pull requests, issues, and code reviews—providing a comprehensive view of developer output over time. Our GitHub data comes from two sources: GHArchive, a public archive of all events on public repositories, and the GitHub REST API, which allows targeted queries for individual developer activity. Using these data sources together with other datasets described above, we first construct a set of treated developers who adopt one of the AI tools that we study. We also construct a set of developers that we use as our control pool, defined as continuously active developers whose first recorded activity occurred before January 2020.

For the set of treated and control developers, we use the GitHub REST API to download their complete commit and pull request histories through March 2026.[11] For each commit, we retrieve the author, timestamp, and commit message; commit messages may contain co-author information in a standardized trailer format, which we parse to identify collaborators. 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.

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 included in the Copilot subscription, including autocompletion, the use of sync coding agents in VS Code and other platforms, and the use of the async coding agent on GitHub, between April and December 2025. We use these data to construct adoption events for different AI coding tools by identifying each user's first observed usage and to analyze

---

[10] See GitHub—Octoverse 2025.

[11] We rely on the GitHub REST API rather than GHArchive because we detected issues in the GHArchive data, including missing dates and incomplete coverage of commit activity, especially after mid-2024. We therefore conclude that GHArchive is useful for identifying active developers but not for measuring their activity.

subsequent usage patterns over time.

In addition, we use subscription data for two GitHub products—GitHub Copilot and GitHub Pro—covering the period from 2021 onward. These data include the start and end dates of each user's subscription and are used to identify the adoption of autocomplete and GitHub Pro.

Marketplace Data We obtain panel data from various providers from March 2020 through May 2026 on four software marketplaces—the Apple App Store, the Google Play Store, the Chrome Web Store, and SourceForge—to analyze changes in the number of apps released over time and the consumption of these new apps. Appendix B.6 provides the full details on data sources, collection method, and variables for each marketplace.

5.2 Construction of Adoption Events

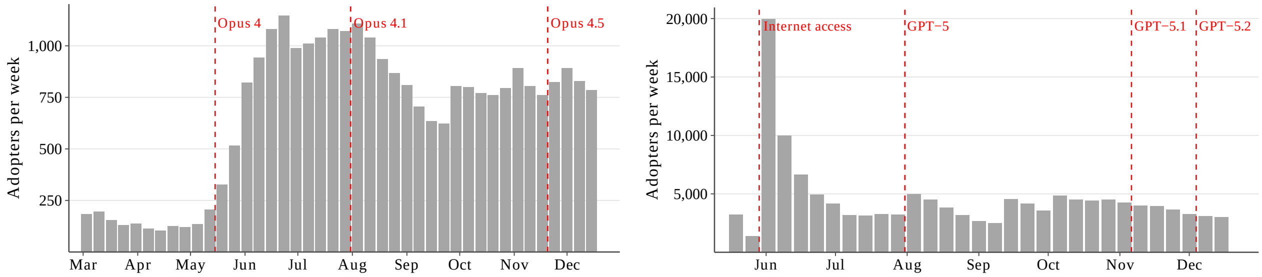
Measuring AI adoption across different tools is challenging because detailed usage data are typically proprietary. While we observe internal usage data for GitHub Copilot, comparable data are not available for other tools. We address this challenge by combining the confidential GitHub Copilot data with publicly observable GitHub activity to identify adoption events across multiple tools and companies.

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. We restrict the sample to 2022 to avoid spillover effects from AI chatbots that became prevalent in 2023.

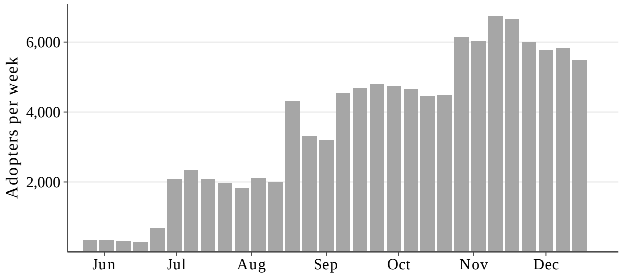
Sync Agents For sync agents, we identify two tools: GitHub Copilot and Claude Code. GitHub Copilot adoption comes from internal Microsoft data, where we use the first date each user invokes GitHub Copilot in agent mode as the adoption date. To isolate sync agent usage from async agent usage, we restrict this sample to users who never use the GitHub Async Agent. We identify Claude Code Sync Agent 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 4(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.

Async Agents By contrast with sync agents, async agents directly interact with the version-tracking system and hence leave unambiguous traces. We identify GitHub Async

Figure 4: Adoption Events over Time



(a) Claude Code **(b)** OpenAI Codex



(c) GitHub Async Agent

Notes: Each panel plots the weekly count of new adopters for the corresponding tool over our sample window, restricted to developers who eventually appear in our matched event-study samples. 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 4(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 4(b). Panel 4(c) omits such markers as the underlying model behind GitHub Async Agent has not been publicly versioned during our window. We omit the corresponding distributions for GitHub Autocomplete and GitHub Sync Agent as those data are considered sensitive.

Agent adoption events using public GitHub data. We retrieve all pull requests created by the Copilot coding agent bot (`copilot-swe-agent[bot]`) and identify the associated human users through co-author information in commit messages, requested reviewers, or assignees. For each user, we define the adoption date as the creation date of their earliest associated Copilot pull request.[12] We identify Codex Async Agent adopters similarly, retrieving all pull requests with the branch prefix `codex/`, which indicates the PR was created by OpenAI's Codex agent. For each user, the adoption date is the creation date of their earliest Codex pull request. As with GitHub Async Agent, we treat this as the adoption of both sync and async coding agents. Figure 4(b) and 4(c) plot the resulting adoption events for the two async tools. While Codex adoption jumps sharply with the

---

[12] Adoption of an async agent does not imply exclusive use of that agent: only a small share of developers use the async agent exclusively, and we confirmed using Microsoft internal data that async agent adoption is associated with roughly a fivefold increase in sync agent usage. We therefore interpret async agent adoption as capturing the joint adoption of async and sync capabilities, and isolate their separate effects by analyzing agent-generated and human-generated commits separately.

introduction of internet access in early June 2025 and remains roughly stable thereafter, GitHub Async 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 systematically—an important caveat when comparing treatment effects across the tools, as cross-tool differences could partly reflect heterogeneity in adopters rather than the tools themselves.

5.3 Sample Construction

Many developers on GitHub are inactive or only sporadically active. Moreover, our initial investigation suggests that many repositories or even accounts are created solely by users to try out coding tools on GitHub, rather than for genuine development. To address these concerns, we impose the restriction that, for any adoption event, the user must have at least one week with positive commits at or before week -11 relative to the adoption date (when our event study starts). This requirement ensures we focus on developers with engagement on the platform before adopting the AI coding tool, rather than users who created accounts or repositories solely to experiment with these tools.

Our sample period varies across the AI tools we study, reflecting differences in when each tool became available. For each tool, we begin the developer sample in the calendar year of its launch and run through the end of that same year: GitHub Autocomplete (May 2022), GitHub Sync Agent (February 2025), Claude Code (February 2025), GitHub Async Agent (May 2025), and Codex (May 2025). 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 information only from the main branches of the repositories to which developers contribute. As a result, activity occurring on secondary or feature branches, as well as any subsequently removed content, is not captured in our dataset.

5.4 Summary Statistics

Table 2 presents pre-adoption summary statistics for adopters across the three main samples. The table reports weekly averages for six outcomes spanning the production hierarchy, from lines changed to releases. Several patterns are worth noting. First, autocomplete adopters are substantially more active than sync and async adopters across all measures: they average roughly 18 commits per week, compared to 6 for the pooled sync and async samples. This likely reflects our requirement that autocomplete adopters have a pull request in the adoption week (as described in Section 6.1), which selects for more active developers but ensures comparable activity in the matched control group. Second, there is substantial heterogeneity in activity across developers within each sample: for commits,

Table 2: Pre-Adoption Summary Statistics for Adopters

Sample	Lines Changed	Distinct Files	Commits	PRs Created	Distinct Repos	Releases
GitHub Autocomplete	42467.77	90.73	17.82	1.59	3.62	1.17
(sd)	(461797.17)	(362.70)	(50.35)	(2.95)	(10.14)	(7.29)
[p5, p95]	[0.00, 114183.20]	[0.00, 306.87]	[0.00, 65.31]	[0.00, 6.00]	[0.00, 13.27]	[0.00, 4.00]
Pooled Sync	25905.54	26.90	5.79	0.64	0.79	0.39
(sd)	(658287.76)	(188.83)	(21.09)	(4.06)	(2.37)	(4.48)
[p5, p95]	[0.00, 68838.17]	[0.00, 108.55]	[0.00, 22.91]	[0.00, 3.09]	[0.00, 2.73]	[0.00, 1.27]
Pooled Async	26994.77	25.67	5.46	0.59	0.74	0.37
(sd)	(717240.67)	(193.66)	(20.18)	(4.17)	(2.06)	(4.69)
[p5, p95]	[0.00, 72505.25]	[0.00, 104.91]	[0.00, 21.55]	[0.00, 2.82]	[0.00, 2.55]	[0.00, 1.09]

Notes: This table reports pre-adoption summary statistics across all outcomes we consider and the three main samples of adopters. "Pooled Sync" pools adopters of GitHub Sync Agent, GitHub Async Agent, Claude Code, and OpenAI Codex, as in the sync event studies. "Pooled Async" pools adopters of GitHub Async 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.

the standard deviation is roughly 3–9 times the mean, reflecting the mix of full-time professional developers and sporadic open-source contributors. Accounting for this variation will be important when we develop our empirical strategy. Third, the number of releases is small relative to other outcomes, reflecting the fact that most developers do not create releases in any given week; we return to this point when interpreting the attenuation of effects across the production hierarchy.

6 Productivity 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. In this section, we first use weekly commits, our primary task-level productivity measure, to estimate the productivity effects of each tool by pooling adopters into the three generations (autocomplete, sync agents, async agents). We then study heterogeneity across users by baseline activity, the productivity effects of individual tools, and how the effect changes over time within a generation. In the next section, we move beyond commits and examine outcomes at each layer of the production hierarchy, from lines of code through releases. We interpret these results through the lens of our production function model to understand how large task-level productivity effects translate into gains in final output.

6.1 Empirical Strategy

For each coding tool, we perform a matched event study, matching users who adopt the tool on a given date with control users who never adopt the tool but are similarly active on that date.[13] This restriction is important: users are more likely to adopt a coding tool during an activity spell, so if we did not impose the same activity requirement on controls, we would violate the parallel trends assumption and overestimate the tool's productivity effect. We implement this strategy by selecting a control developer who made a pull request on the adoption date of the matched treated user.[14]

When matching control users' outcomes to those of treated users, we use control group outcomes from exactly one year before the corresponding treated group outcomes. We make this choice for two reasons. First, we are concerned about widespread private adoption of generative AI coding tools: while we can confirm that our treated group adopts, other users may also adopt these tools without leaving a trace in public repositories. A contemporaneous control group may therefore be contaminated by treatment effects. Second, even in the absence of contamination, adopters may be positively selected on unobservable characteristics relative to non-adopters. As a result, selecting control developers among non-adopters in the same year may lead us to compare treated developers with otherwise less active developers, which would generate a spurious treatment effect. By drawing controls from the previous year, we avoid this negative selection in the donor pool.

Using this matched sample, we estimate a generalized difference-in-differences event study:

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

where Y_{it}^{T} is the weekly outcome of treated developer i in week t, normalized by her pre-period mean[15]; Y_{it}^{C} is the analogous outcome for the matched control; t_i^{*} is the (pseudo-)adoption date; and α_i is a matched-pair fixed effect. We normalize the pre-period co-

---

[13] Adoption coincides with observable activity for some tools but not others. For Codex Async Agent, GitHub Async Agent, and Claude Code Sync Agent, we identify adoption through pull requests or file creations that appear directly in our data. In contrast, for GitHub Autocomplete and GitHub Sync Agent, we identify adoption through internal subscription or usage data that does not necessarily correspond to public GitHub activity. To ensure a uniform structure across all tools, we require that developers have at least one pull request and one commit during the week of adoption for GitHub Autocomplete and GitHub Sync Agent, and we similarly strengthen our requirements for the Control group to also include a commit for these two event studies only.

[14] Even given our requirement that the control group be active on the date of adoption, one may worry that pull requests serve as a weaker signal of future activity than the adoption of a new coding tool (which may have been adopted in anticipation of a future period of high activity). To address this concern, we explore the effects of adopting two non-AI coding tools in our robustness section below. Reassuringly, we find very small effects from adopting these placebo tools.

[15] We drop developers with no commits in the pre-period.

efficients 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. We estimate this specification for several outcome measures, including lines of code, distinct files, commits, pull requests, distinct repositories, and releases; in all specifications, we winsorize the dependent variable at the 1st and 99th percentiles. Appendix C gives the full specification, including pre- and post-window indicators.

Because we normalize the treated and control outcomes by each developer's pre-period mean, the coefficients β_k should be interpreted as the average of individual percentage changes, not a renormalization of the average level effect relative to average productivity—a distinction that matters if treatment effects are heterogeneous across baseline productivity levels, which is likely the case here, given the wide range of baseline activity in our sample, from hundreds of commits per week to a single one.[16] The interpretation of β_k is identical to that of treatment effects on log-transformed outcomes; we avoid log-transforming because maintaining a balanced panel would then require dropping users with zero commits in any week, which is problematic given the sporadic activity of many users in our sample. We prefer normalized effects because they capture the effect experienced by the average user and because outcomes in levels exhibit high variability that makes precise estimation difficult.[17]

We report summary statistics comparing treated and matched control units for each event study in Table 3. Because we match on commits, the gap in pre-period commit activity between treated and control groups is substantially narrowed, though some differences remain. Remaining differences in other outcomes (lines changed, files, pull requests) are expected given that we do not match on these variables directly; we verify below that pre-trends are flat for all outcomes, supporting the parallel trends assumption. We also note that the sample for releases is substantially smaller than for other outcomes: because our normalized outcome requires dividing by the pre-period mean, we drop developers with no releases in the pre-period, and releases are rare enough that this excludes the majority of developers.

---

[16] For example, suppose treatment increases the output of less productive developers more. A developer who averages 1 commit per week increases to 2 (a 100% gain), while one who averages 100 commits increases to 110 (a 10% gain). The average percentage change is 55%. But the average level change is 5.5 commits, which expressed as a percentage of the overall mean (101 commits) would only yield an effect of 5%.

[17] A potential concern is that the pre-period mean used to normalize the outcomes is estimated from only 11 weeks and is therefore noisy. By Jensen's inequality, dividing each weekly outcome by this noisy pre-period mean introduces a small upward bias of second order in the noise variance. To address this, we re-estimate the same specification using a longer 52-week pre-period to compute the mean and find that the bias is very small.

Table 3: Comparison of Treated and Matched Control Units

	GitHub Autocomplete	GitHub Sync Agent	Claude Code Sync Agent	GitHub Async Agent	Codex Async Agent	GitHub Pro	Dockerfile
Median Weekly Lines Changed							
# Treated Units	13,899	5,123	17,180	56,658	40,523	1,177	39,065
Treated Group	2,163	1,003	1,618	1,552	1,165	1,863	920
Matched Control Group	838	475	468	458	286	822	246
Median Weekly Distinct Files							
# Treated Units	13,914	5,137	17,212	56,852	40,742	1,178	39,216
Treated Group	17.82	10.64	11.82	11.45	7.73	15.18	7.64
Matched Control Group	11.55	7.00	6.91	6.36	4.55	10.55	4.00
Median # Weekly Commits							
# Treated Units	14,026	5,186	17,346	57,377	41,139	1,183	39,854
Treated Group	4.45	2.64	3.00	2.64	1.82	4.36	1.73
Matched Control Group	4.09	2.64	2.36	2.09	1.45	3.73	1.36
Donor Pool	3.67	3.07	3.00	3.00	3.00	3.67	3.00
Median Weekly PRs Created							
# Treated Units	10,427	3,494	8,242	25,214	12,198	663	10,953
Treated Group	1.09	0.91	0.55	0.64	0.45	1.00	0.36
Matched Control Group	0.91	0.64	0.91	0.73	0.73	0.91	0.64
Median Weekly Distinct Repos							
# Treated Units	13,920	5,140	17,221	56,877	40,771	1,180	39,233
Treated Group	1.00	0.64	0.64	0.55	0.45	0.86	0.45
Matched Control Group	1.00	0.73	0.64	0.64	0.55	1.00	0.45
Median Weekly Releases							
# Treated Units	3,522	964	2,800	7,335	3,033	273	3,566
Treated Group	0.73	0.64	0.55	0.55	0.36	0.55	0.45
Matched Control Group	0.45	0.55	0.55	0.55	0.55	0.64	0.45
Sample Period Start Date	2022-06-21	2025-05-13	2025-02-24	2025-05-19	2025-05-16	2022-06-21	2025-01-01
Sample Period End Date	2022-12-26	2025-12-23	2025-12-22	2025-12-22	2025-12-22	2022-12-26	2025-12-22
Size of Donor Pool	2,316,931	2,255,660	2,226,187	2,219,113	2,227,512	2,324,449	2,199,262

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 specific regression after all sample filters (pre-period zero removal, upper-bound check, mean-zero/NA removal for average outcomes). "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. The Sample Period refers to the time when outcomes for the treated group are measured; the Control group outcomes are measured one calendar year earlier (see discussion in Empirical Strategy section).

To estimate the effects of different tools individually, we pool observations across products in the main analysis and report product-specific effects in a subsequent heterogeneity analysis. For autocomplete, we observe only one product—GitHub Autocomplete—and therefore restrict the sample to users of that tool. The analysis of sync and async agents is more complex. While we observe two products that operate purely as sync agents, adoption of async agents reflects the use of both async and sync functionalities, as discussed above. To separately identify the two effects, we pool users of both async agents and decompose their activity into an agent-authored component and a human-authored component. The agent-authored component identifies the async-agent effect, since we directly observe the agent's output. The human-authored component, pooled with sync-agent users, identifies the sync-agent effect. The idea is that async agents can increase observable output only through their own contributions, so any change in human-authored activity must come from sync agents.

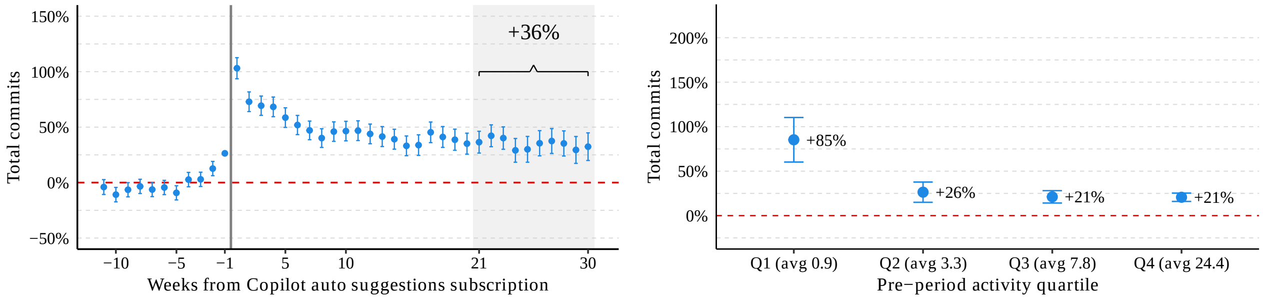
6.2 Productivity Effects Across Generations

In this section, we present results on how task-level productivity differs across generations of AI tools over time. 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 other 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 7, we examine other outcome variables to understand how AI affects different stages of the production process and final output.

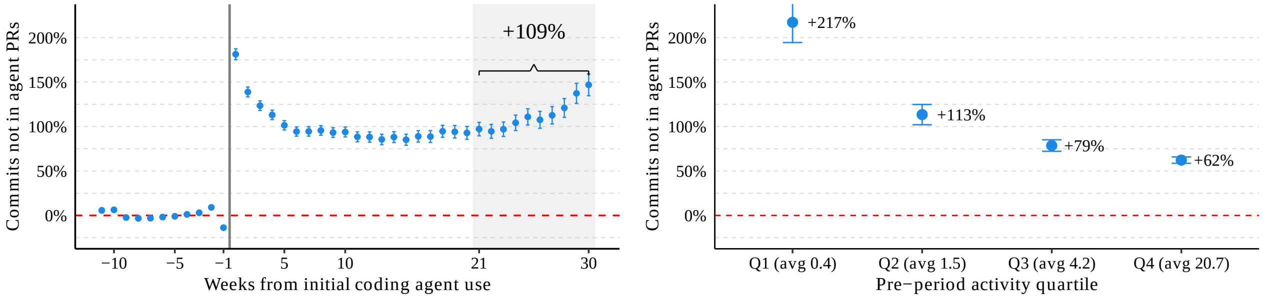
We present our event studies in Figure 5 and the corresponding estimates from a regression specification that reports effects separately for weeks 1–10, 11–20, and 21–30 after adoption in Table 4. The effect is shown separately for each generation: autocomplete (all commits), sync agents (human-authored commits), and async agents (agent-authored commits). The pre-period coefficients are normalized to have zero mean.

The top row shows users who adopted GitHub Copilot in 2022, when only AI-based code completion was available. Our estimate in Figure 5(a) shows a 36% increase in the number of commits. The pre-trend is almost perfectly flat, supporting the parallel trends assumption. As shown in Figure 5(b), the effect is larger for less active developers (85% in

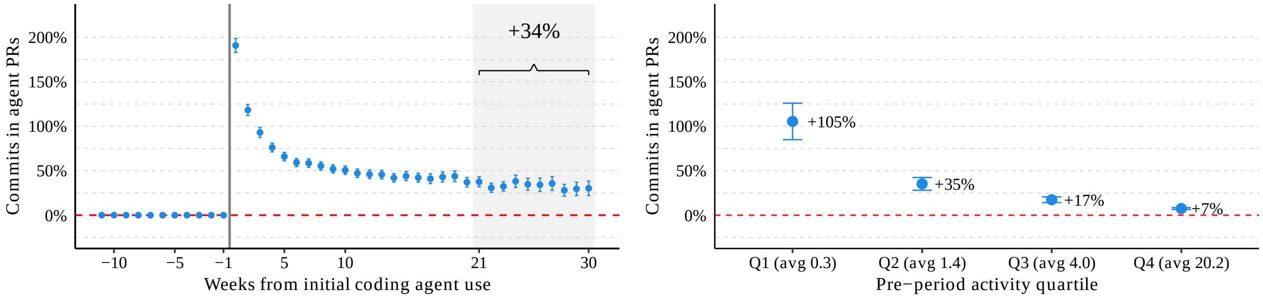
Figure 5: Productivity Effects of GenAI Coding Tools



(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 5(a) and 5(c)) or interrupted time series (Panel 5(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 5(a) compares users who subscribed to GitHub Copilot in 2022 (when it provided only autocomplete) to a matched control group active a year prior. Panel 5(c) pools adopters of sync agents (Claude Code, GitHub Sync, GitHub Agent, and OpenAI Codex) against matched controls. Panel 5(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 87%, 259%, and 554% for Panels 5(a)–5(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.

the least active quartile), though it remains substantial throughout the activity distribution (from 21% for the most active to 85% for the least active). Our estimates are consistent with prior field-experimental estimates of the effects of GitHub Copilot autocomplete (Cui et al., 2026), providing external validation of our matched event-study design.

Moving on to sync agents in the middle row, we again find remarkably flat pre-trends and a sizable effect immediately upon adoption (Figure 5(c)). While the number of commits temporarily increases sharply—likely due to an experimentation phase—the effect rapidly decays before stabilizing at a 110% productivity gain—about triple the effect of autocomplete. The heterogeneity pattern (Figure 5(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 62%.

Finally, the bottom row shows the effect of adopting async agents on agent-written commits (Figure 5(e)). Agent-authored commits are zero by definition before adoption, and they increase by about 30% of a user's baseline commits after adoption. As with the other tools, we see large short-run effects—consistent with an initial experimentation phase in which developers learn how to use the tool effectively—that settle down to more moderate but still significant levels, with effects again larger for less active developers (Figure 5(f)).

In Table 4, we break out the sync and async effects by product; the corresponding per-tool event studies are in Appendix Figure OA-3. 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 199% for Claude Code, 43% for GitHub Sync Agent, and 94% 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 shows a notably larger sync effect than the other tools, which can be attributed to the fact that Claude Code is primarily operated via a command-line interface, and its adopters may be more sophisticated coders who would benefit more from any tool. Consistent with this, Figure 4 shows that Claude Code and Codex adoption is concentrated earlier in our sample. Our empirical strategy cannot distinguish between these mechanisms generating heterogeneity across tools.

Taken together, our results establish that generative AI coding tools have substantial effects on developer productivity, with effects that grow across successive generations. Autocomplete increases commits by roughly 40%; the cumulative effect including sync agents rises to approximately 140%; and additional use of async agents raises it to 180% through agent-authored commits. These effects are stable across tools within each cate-

Table 4: Change in Number of Commits by Coding Tool

	% Change in Number of Commits		
	Weeks 1–10	Weeks 11–20	Weeks 21–30
Autocomplete	61.3	40.3	35.9
	(2.4)	(2.5)	(2.9)
Sync Effect	115.3	89.7	109.1
	(1.6)	(1.8)	(2.7)
Claude Code	186.2	139.5	199.2
	(4.7)	(4.9)	(7.2)
GitHub Sync	106.9	51.0	42.7
	(5.9)	(5.7)	(6.9)
OpenAI Codex	106.5	74.3	94.3
	(3.2)	(3.5)	(4.6)
Async Effect	85.2	43.6	33.6
	(1.8)	(1.7)	(2.1)
GitHub Agent	52.8	36.8	41.9
	(1.4)	(1.7)	(2.9)
OpenAI Codex	134.5	51.4	31.0
	(4.1)	(3.2)	(3.1)

Notes: This table reports results that pool event-study coefficients measuring the effect of the adoption of various Generative AI coding tools on software developers' productivity at various time horizons. We caution that the estimates in the column 'Weeks 1-10' may be confounded by temporary experimentation or activity bias, which is why our preferred estimates are those corresponding to the long-run effects measured in column "Weeks 21-30".

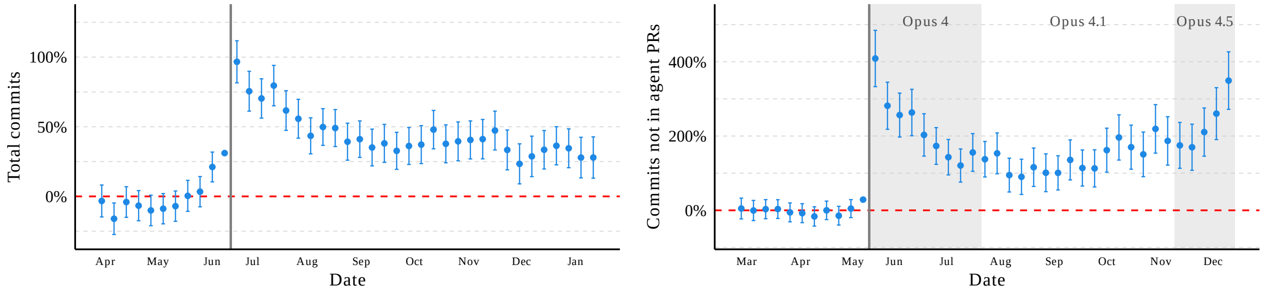
gory, and persist over at least 30 weeks. The significantly smaller effects of autocomplete relative to agents are consistent with the nature of these tools as discussed in Section 4: autocomplete simply augments the developer's own input when writing code, whereas agents can lead to partial or full automation.

6.3 Dynamic Treatment Effects

The treatment effects in Figure 5(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 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 6 reports the results.

Panel 6(b) shows that the increasing event-time pattern for Claude Code is preserved

Figure 6: Productivity Effects over Calendar Time



(a) GitHub Copilot 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. Restricting attention to a narrow adoption window means that event time maps directly onto calendar time, so the horizontal axis can be labelled by calendar date rather than weeks since adoption. The empirical specification is otherwise identical to the matched event study in Figure 5(c). Panel 6(a) shows GitHub Copilot 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 6(b) shows Claude Code adopters in mid-2025; 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.

within the early-adopter cohort, and aligns naturally 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 6(a) reinforces this reading by contrast: GitHub Copilot Autocomplete showed no increasing pattern in event time (Figure 5(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, both 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.[18]

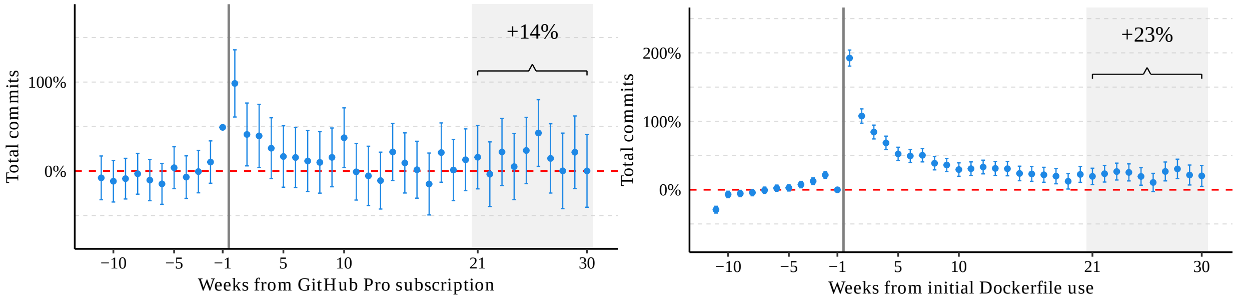
6.4 Robustness Checks

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

---

[18] 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.

Figure 7: 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. We focus on users who were active at least 12 weeks prior to the event date. Panel 7(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 a year prior on the same day. Panel 7(b) compares users who adopted Docker in 2025 to a matched control group of users who authored a pull request a year prior on the same day. 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 137% and 413% for Panels 7(a) and 7(b), respectively.

Activity bias. A central 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 interpretation. First, we examine the adoption of two non-AI coding tools: GitHub Pro (a paid subscription without AI features, used as a placebo) and Docker (a containerization tool identified in a similar way to Claude Code, used as an upper bound since Docker may itself raise productivity). Figure 7 shows that both tools generate large short-run spikes in activity that decay rapidly: GitHub Pro converges to no long-run effect, while Docker stabilizes around 23%. AI coding tools, by contrast, stabilize at productivity gains of around 109%. Even taking the Docker effect as an upper bound on activity bias, our AI estimates remain large. Second, if our estimates were driven by activity bias, we would expect them to dissipate over time, as they do in the placebo exercises (within roughly five weeks). Instead, as seen in Table 4, the AI treatment effects remain stable over the 30-week post-adoption window—settling down for async agents and strengthening for sync agents.

Empirical specification. To assess the robustness of our specification that scales outcomes by the pre-period mean, we also estimate a matching specification 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-5 show that this alternative approach yields effects closely matching our pooled normalized estimates.

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. 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. Our sample is therefore an undercount, but the public activity of identified adopters captures the large majority of their workflows, supporting a representative picture for our productivity estimates.

7 Productivity 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 due to the weak-link hypothesis, 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.

7.1 How Does Task-Level Productivity Affect Final Output?

In Table 5, 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 and commits, and smaller still for pull requests, distinct repositories, and releases. Starting with autocomplete, the effects decline monotonically across layers: from 228.2% for lines of code, to 35.9% for commits, and to only 10.2% for releases. This is precisely the pattern predicted by Corollary 1. Because autocomplete operates only at the code-writing layer (Section 4), any upstream gain must propagate through every higher layer of human-bottlenecked aggregation, and the model predicts that each such layer absorbs a fraction of the upstream gain.

Moving to sync agents, the same monotonic pattern holds at larger magnitudes: the effect on lines of code is 741.3%, falling to 25.5% for distinct repositories touched and 20.2% for releases. The decay across layers looks similar when we examine individual sync tools rather than the pooled effect. Furthermore, the release effect for sync agents is larger than for autocomplete. It is worth noting that the model does not necessarily imply

Table 5: Attenuation of Productivity Effects Across Production Layers

	Outcomes (Weeks 21–30, %)					
	Lines Chg	Dist Files	Commits	PRs	Dist Repos	Releases
Autocomplete	228.2	50.8	35.9	11.0	13.6	10.2
	(17.6)	(4.5)	(2.9)	(2.7)	(1.7)	(4.1)
Sync Effect	741.3	187.0	109.1	65.5	25.5	20.3
	(19.2)	(4.3)	(2.7)	(2.3)	(1.0)	(2.6)
Claude Code	981.2	288.4	199.2	145.5	49.2	29.2
	(47.3)	(11.3)	(7.2)	(7.1)	(2.3)	(6.0)
GitHub Sync	470.9	106.2	42.7	21.4	11.3	1.3
	(47.0)	(11.7)	(6.9)	(6.0)	(3.4)	(8.2)
OpenAI Codex	792.6	186.4	94.3	61.8	14.6	0.2
	(36.1)	(7.5)	(4.6)	(4.4)	(1.6)	(4.3)
Async Effect	658.3	52.4	33.6	71.8	13.8	—
	(74.7)	(3.7)	(2.1)	(5.6)	(0.4)	
GitHub Agent	879.3	63.0	41.9	66.6	18.9	—
	(141.7)	(5.7)	(2.9)	(5.3)	(0.7)	
OpenAI Codex	549.8	47.9	31.0	80.0	11.5	—
	(86.2)	(4.8)	(3.1)	(10.3)	(0.5)	

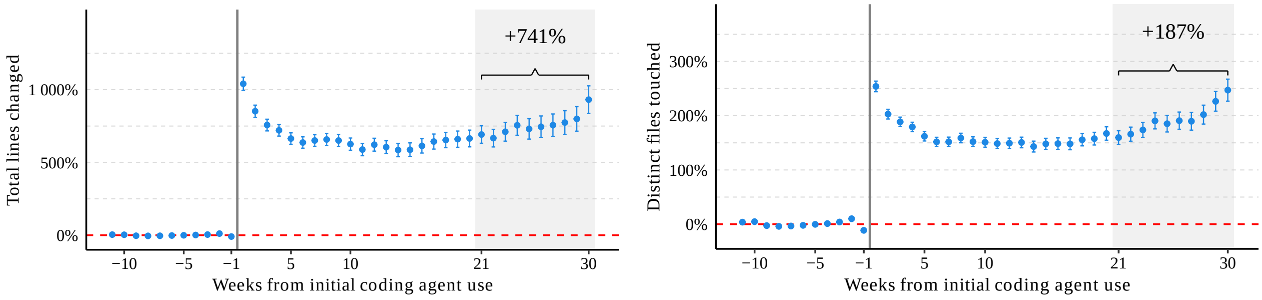
Notes: This table reports event-study coefficients (1% winsorized, weeks 21–30 bin, normalized by pre-period mean) measuring the effect of adoption of various Generative AI coding tools on outcomes spanning the production hierarchy. Lines Chg = weekly total lines changed, Dist Files = distinct files, Commits = weekly commits, PRs = weekly PRs created, Dist Repos = distinct repositories contributed to, Releases = weekly repo release count. Releases are not reported for async tools because agent-authored commits are attributed to the developer's repositories, making it difficult to isolate the agent's contribution to releases.

a strictly decreasing pattern for sync agents, since they enter at multiple layers (Section 4): if they are more effective at higher layers, the effect size can in principle increase rather than decrease across layers.

Finally, for async agents, we do not report an async release effect because async agents do not have the capability to release a repository directly, so this outcome is zero by construction. Monotonicity holds across the remaining outcomes but is not perfect: the effect on pull requests is 71.8%, substantially larger than the 33.6% effect on commits and 52.4% on files. 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.

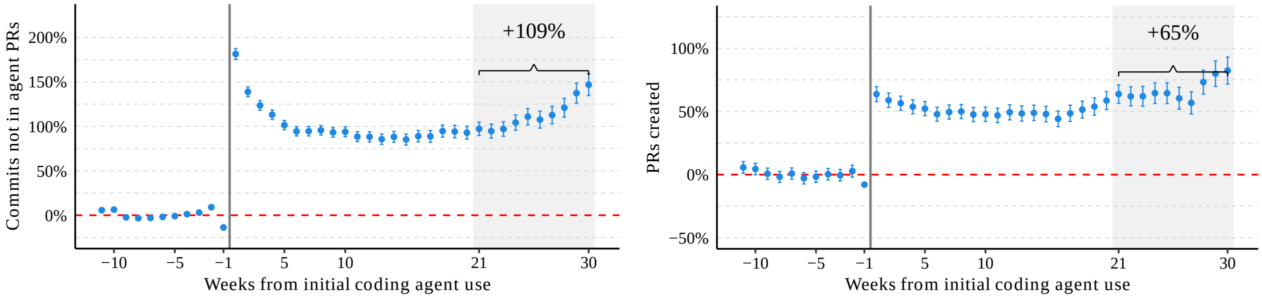
Figure 8 presents the corresponding event studies for the sync effect across all outcomes; the analogous per-tool event studies across all outcomes are reported in Figures OA-4–OA-10 in the Appendix. Several features are worth noting. First, pre-trends are flat for all outcomes, supporting the parallel trends assumption. Second, the effects on lines of code, files, and commits are stable from roughly week 5 onward, consistent with the patterns in the main commit-level event studies. Third, the result for releases is the least visually

Figure 8: Sync Effect Across Production Layers



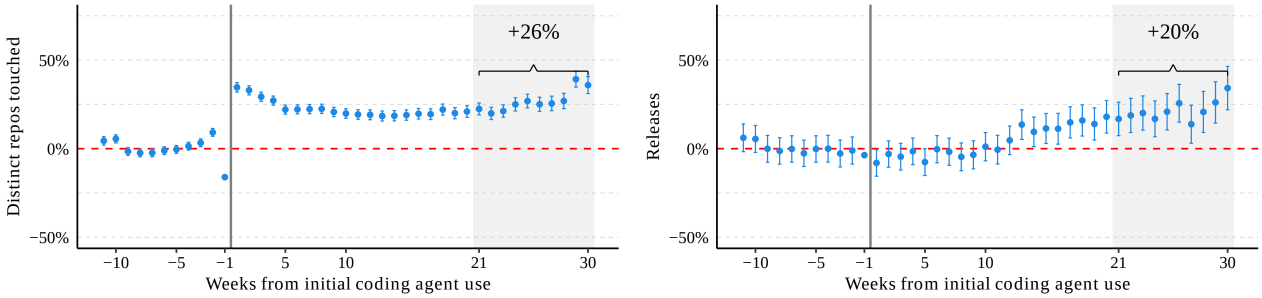
(a) Lines Changed

(b) Distinct Files



(c) Commits

(d) Pull Requests Created



(e) Distinct Repositories

(f) Releases

Notes: This figure presents matched event-study estimates of the sync effect across production layers, corresponding to the columns of Table 5. 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 1,869.9%, 413.4%, 259.0%, −142.5%, 20.7%, and −24.9% for Panels 8(a)–8(f), respectively.

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.

To understand the robustness of these results, we use the placebo analysis and the alternative scaling specification described in Section 6.4. The placebo analysis (Table OA-4) reports the same outcomes across production layers for non-AI coding tools (GitHub Pro and Docker). While some attenuation is visible for these tools, the effects at every layer are much smaller than those of AI coding tools, reinforcing that the gradient we document is not an artifact of our research design. We also estimate the average percentage effect via decile-specific level regressions normalized by each decile's treated pre-mean (Table OA-5). The attenuation pattern across production layers is robust across the baseline-activity distribution.

An alternative interpretation of these results is that the underlying productive output is unchanged but AI's coding style inflates our measures. For example, coding agents are known to write more verbose code, which would mechanically increase lines and files without raising substantive output. While this concern may have some merit for lower-level outcomes (lines of code, distinct files), it does not apply to higher-level outcomes such as commits, pull requests, and releases, where the unit of measurement is not affected by verbosity. Moreover, the concern does not apply to autocomplete, for which the developer accepts or rejects each suggestion—and we still see a clear pattern of attenuation across layers. A related concern is that AI adoption may change the granularity of the units we count: if AI lowers the friction of committing or opening a PR, treated developers might produce more (but smaller) commits or PRs for a given amount of underlying work. This would work against our finding, since mechanically inflating the higher-layer counts would reduce the attenuation we document. Furthermore, average lines-per-commit does not decrease after adoption, indicating that the cross-layer attenuation is not driven by treated developers breaking their work into smaller commits.

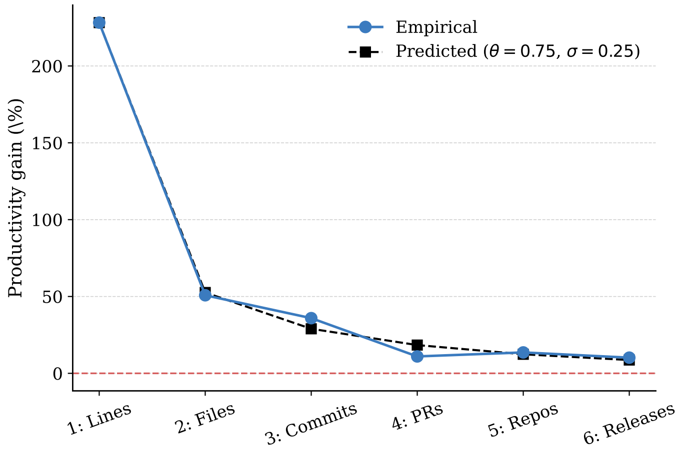
7.2 Quantifying Complementarities in the Production Function

The previous section establishes that productivity gains attenuate across layers, consistent with our layered production function. We now turn to the structural parameters that drive this attenuation. In Section 4, we showed that the elasticity of substitution between upstream and effective input is the key parameter that determines the shape of attenuation across layers. In this section we take a further step and use our empirical results to infer complementarities in the software-development production function by backing out 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.

Proposition 1 showed that pass-through at each layer is governed by two parameters: θ, the output elasticity of each layer's output with respect to upstream input, and σ, the elasticity of substitution between upstream input 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

Figure 9: Calibrated Model Fit for Autocomplete



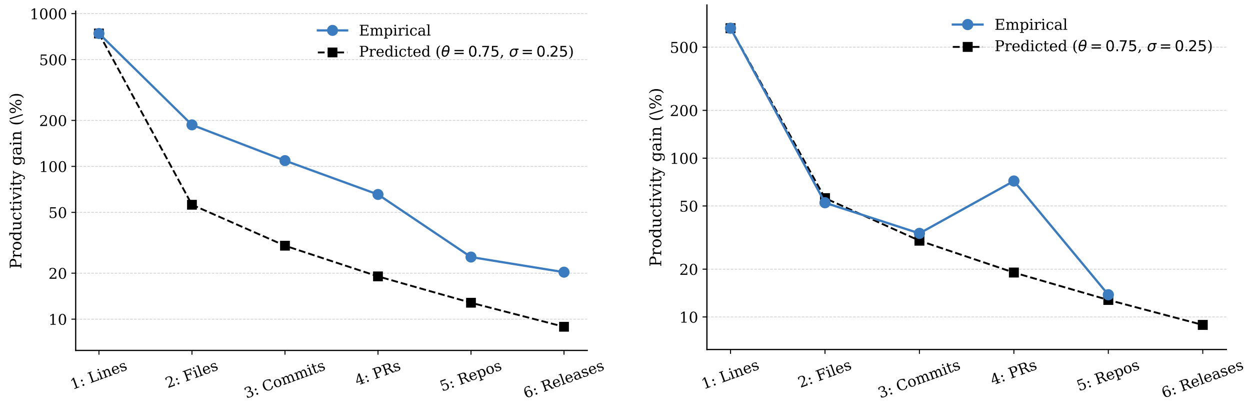
Notes: This figure shows the best-fit model to the autocomplete attenuation pattern (calibrated $\theta = 0.75$, $\sigma = 0.25$). Black squares show the empirical effects at each layer; the colored line shows the model prediction.

enters the production hierarchy at a single layer—code writing—so the model gives a clean prediction: take the empirical lines-of-code effect as the layer-1 shock A, then recursively apply the pass-through formula to obtain predicted gains at each higher layer. Sync and async agents, in contrast, intervene at multiple layers (Section 4), 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 layers 2–6. Appendix A.6 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 curvature of the decay: when $\sigma < 1$ (complements), most of the loss is concentrated at the first layer above the shock, and the log-gain sequence is convex in the layer index; when $\sigma > 1$ (substitutes), losses are spread more evenly, and the log-gain sequence is concave (Proposition 2). The overall steepness of the empirical decay therefore identifies θ, and the curvature identifies σ.

Figure 9 shows the empirical autocomplete attenuation pattern and the model prediction at the best-fit parameters. The best-fit parameters are $\theta \approx 0.75$ and $\sigma \approx 0.25$. The estimated elasticity of substitution lies well below one, placing each layer's production firmly in the complements region: upstream output and effective input must be combined

Figure 10: Predictions for Sync and Async Agents



(a) Sync: Prediction vs. Empirical

(b) Async: Prediction vs. Empirical

Notes: Each panel uses the autocomplete-calibrated $(\theta, \sigma) = (0.75, 0.25)$ 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. Black squares show the empirical effects at each layer; colored lines show model predictions.

in roughly fixed proportions, which is what generates the steep, front-loaded attenuation we observe. The weight on upstream input is $\theta \approx 0.75$, implying that at each layer the upstream output has an output elasticity of roughly 75% and human input at the layer has an output elasticity of roughly 25%.

We next construct a counterfactual prediction for sync and async agents under the assumption that these tools affect only lines of code (layer 1), based on the attenuation pattern obtained from autocomplete. 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 calibrated (θ, σ). Figure 10 compares these counterfactual predictions with the empirical effects. For sync agents, the empirical effects exceed the predicted gains at every intermediate layer; as it was fitted on autocomplete which only affects production of lines of code, the model underpredicts files, commits, and pull requests by roughly two-thirds. For async agents, the empirical commit effect aligns closely with the model prediction, but the empirical effect on pull requests is sharply above the predicted value. As predicted by our discussion earlier in this section, 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.

7.3 Discussion

These results provide direct empirical evidence for the weak-link hypothesis that has been widely discussed in the macroeconomic productivity literature (Jones, 2011; Aghion et al., 2019; Jones, 2026). The pattern we document—large effects at the code-writing layer, attenuating as we move toward final output—is consistent with the model's prediction that human-bottlenecked layers compress upstream productivity gains.

At the same time, we observe that the bottleneck is moving further 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. This suggests that while our current estimates show that productivity effects on shipped software are smaller than the effects measured at the task level, this gap may narrow over time. As AI capabilities improve and tools partially or fully automate higher layers of the production hierarchy, the pattern of attenuation can compress, delivering larger gains to final output.

8 Aggregate Evidence: New Apps and Usage in Software Marketplaces

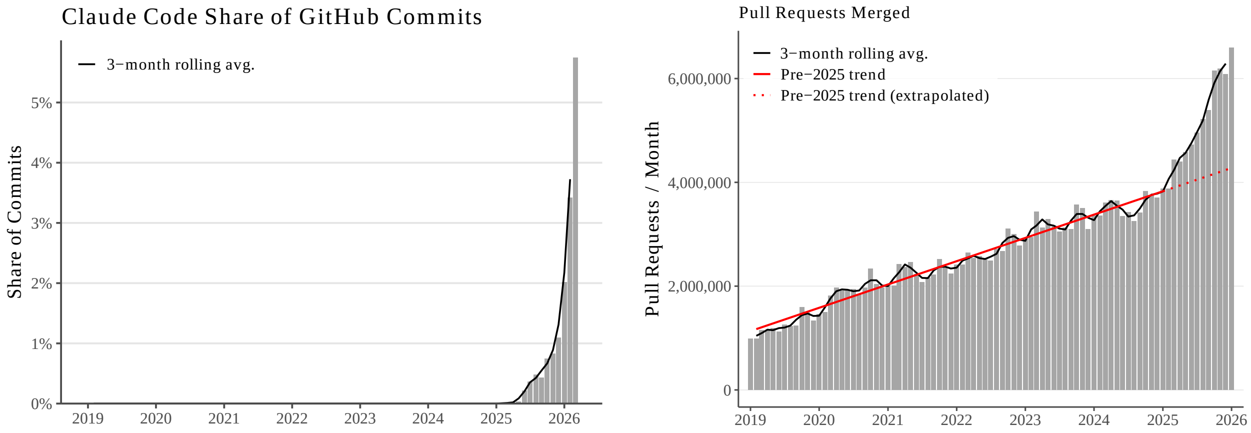
Our developer-level results document substantial productivity gains from AI coding tools, with effects that attenuate sharply across the production hierarchy. A natural question is whether these micro-level effects translate into aggregate changes in software production. This question echoes a long-standing 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).

We examine these questions 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 applications being created and, in particular, consumed by end users.

8.1 Aggregate Coding Activity on GitHub

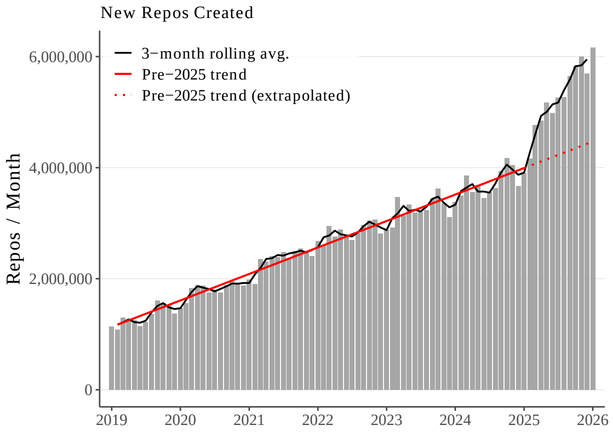
We begin by establishing that aggregate adoption of AI coding tools on GitHub is large enough to make our developer-level effects potentially detectable in platform-wide totals. Figure 11(a) plots the share of commits on public repositories that are signed or authored by just one agentic coding tool—Claude Code. The share has risen rapidly since the tool's release in early 2025 and by early 2026 exceeds 5% of all public commits. Since developers frequently commit work done by AI tools under their own name, and since developers use a plethora of coding tools as examined above, this is almost certainly an undercount of total AI-assisted coding activity. The scale of adoption is therefore consistent with the possibility that developer-level gains could be visible in aggregate platform activity.

Figure 11: 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 early 2026, using data from the GitHub Search API. Bars show monthly counts; the solid line is a rolling average. Claude Code commit share is the fraction of commits signed or authored by Claude Code; developers may commit work done by Claude themselves, so this is likely an undercount. Pull requests merged counts all merged PRs on public repositories. New repos counts non-fork public repositories created.

Figures 11(b) and 11(c) turn to that aggregate activity directly, plotting monthly counts of merged pull requests and newly created (non-fork) public repositories from 2019 through early 2026. Both metrics display a broadly linear trend from 2019 through 2024, reflecting steady organic growth in platform activity. Around early 2025, this trend breaks visibly: both outcomes exhibit a clear acceleration relative to the pre-2025 trend, with timing that aligns closely with the rise in the Claude Code commit share documented above. The acceleration in merged pull requests is also visibly larger than the acceleration in new repositories created, consistent with our developer-level results: new repositories sit at a higher layer of the production hierarchy than pull requests (Section 7), and effects on higher layers should attenuate. Together, these patterns indicate that aggregate coding activity on GitHub has accelerated sharply since early 2025, consistent with both the individual-level productivity gains and the cross-layer attenuation pattern we document.

8.2 Software Marketplaces

The aggregate GitHub evidence shows that coding activity has accelerated since early 2025. A natural question is whether this acceleration is also visible on the consumer side: are more applications reaching end users, and are end users consuming more software? In contrast to aggregate productivity in the broader economy, software output is reasonably observable: application marketplaces record when new applications 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.

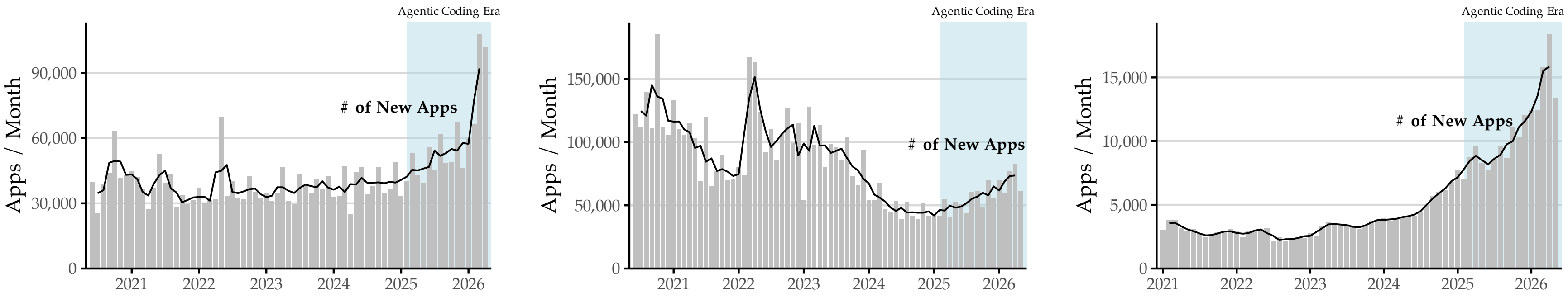
8.2.1 Data and Measurement

We assemble monthly panels for four of the largest application 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, particularly for Linux and Windows). Together these four span different developer populations and provide a useful cross-section for assessing whether the developer-level productivity gains we document are visible in aggregate software output.

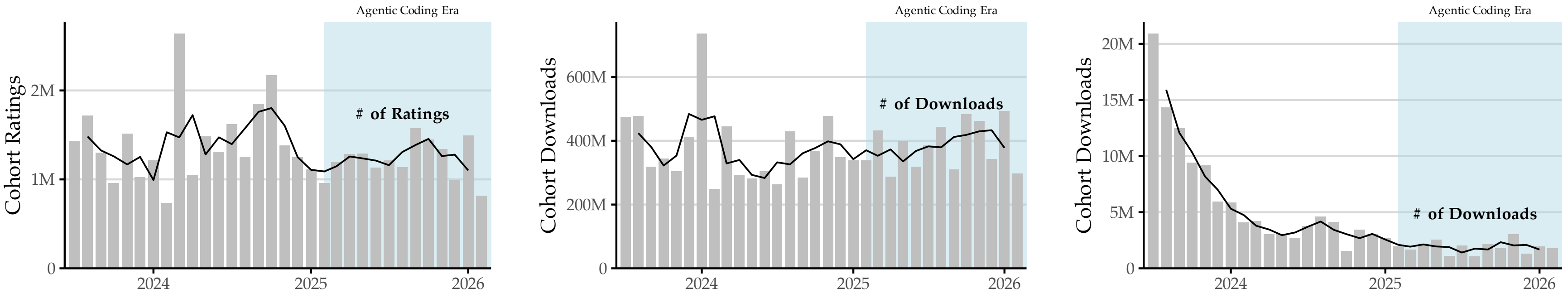
For each marketplace we observe each application's first-seen date and (for all but SourceForge) cumulative end-user usage thereafter. The usage measure is the rating count for iOS (where we do not directly observe installs) and the cumulative install count for Android and Chrome. Because applications can exit the catalog, we use a panel of repeated snapshots rather than a single cross-section of currently listed apps, which would undercount earlier cohorts. We also drop applications owned by large platform operators (Apple, Google, Microsoft, Samsung), which are likely to be pre-installed and whose activity is unlikely to be informative about marginal developer productivity.

Each monthly cohort consists of the non-incumbent applications that first appeared in that month. 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.

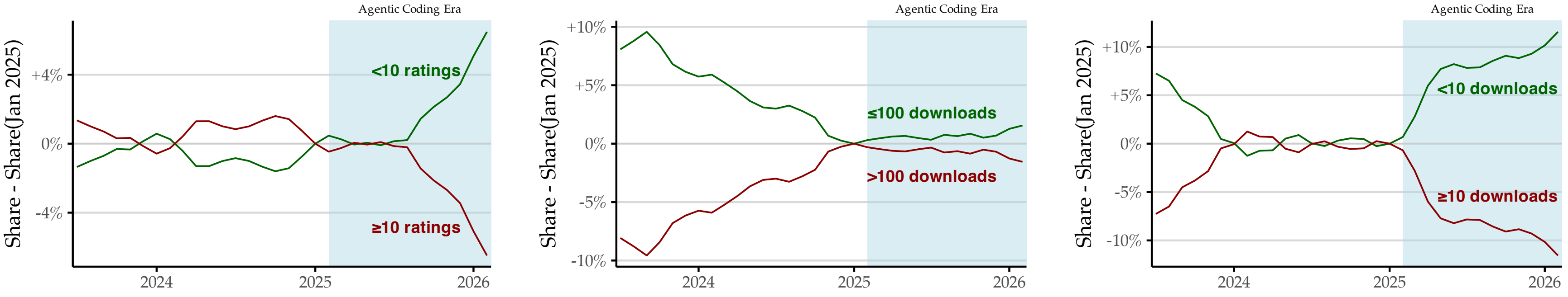
Figure 12: New Apps and Their Usage Across App Stores



(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 applications, 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).

8.2.2 *Releases of New Applications*

The top row of Figure 12 plots monthly counts of new releases for each marketplace. To help the reader assess whether agentic coding accelerated new releases, we highlight the period from February 2025 onward as the agentic-coding era on each plot.[19]

The Apple App Store shows the clearest acceleration. New iOS applications run between roughly 30,000 and 50,000 per month from 2023 through early 2025, then rise sharply over the course of 2025, reaching approximately 100,000 per month by April 2026. The acceleration also gathers pace as we enter 2026, consistent with the Claude Code effect rising over time as the underlying model improves (Figure 6(b)).

The Google Play Store shows a different pattern. Monthly new releases had been declining steadily from over 120,000 in mid-2020 to about 42,000 in January 2025, and the trajectory then breaks: releases stabilize and recover modestly through the end of our sample, reaching about 60,000 per month by mid-2026.[20]

The Chrome Web Store shows a slower but visible acceleration: new extensions roughly double from around 5,000 per month in 2023 to about 13,000 per month by mid-2026. The acceleration also starts earlier than for iOS; growth picks up in 2025 but the trend already breaks from a stable period in 2024. Chrome extensions are simpler artifacts than mobile apps, so tools that predate agentic coding, such as autocomplete, may already have raised production rates earlier.

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. By contrast, the software marketplace SourceForge shows no acceleration (Appendix Figure OA-11), consistent with a developer base that is using AI less.

8.2.3 *Consumption*

We now investigate whether the increase in new releases translates into additional consumption and consumer surplus.

We first examine total cohort-level usage. The middle row of Figure 12 plots, for each entry month, the sum of usage over the next three months across all applications in that cohort. If the additional applications had generated meaningful consumption, the aggregate would rise with the size of the cohort. Instead, the aggregate is flat or declining in all three marketplaces: iOS cohort ratings and Android cohort downloads are both

---

[19] We choose this date because GitHub launched Copilot Agent Mode, the first widely used agentic coding product, in late February 2025; other agentic tools followed soon after. See `https://code.visualstudio.com/blogs/2025/02/24/introducing-copilot-agent-mode`.

[20] Google Play removes substantially more applications than the other stores, which makes the underlying series noisier and harder to interpret.

roughly stable when comparing 2024 to 2025, and Chrome cohort downloads fall rapidly but had already been falling in the pre-period.

The fact that aggregate usage is flat or declining 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 applications 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 application's user base would be small but the aggregate would rise with the number of new releases. While this channel has been documented for AI-written books on Amazon (Reimers and Waldfogel, 2026), it is not visible in the software marketplaces we study.

Still, the fact that aggregate usage does not increase is not enough to conclude that consumer surplus is unaffected, as the introduction of additional apps could have improved *matching* between consumers and applications. From a consumer's perspective, applications are partial substitutes, since time spent on any one mobile app is bounded by total leisure time, so an additional new app need not raise total usage. The new applications 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 application as failing to reach an audience if it accumulates fewer than 10 ratings (iOS), 100 downloads (Android), or 10 downloads (Chrome) in its first three months.[21] The bottom row of Figure 12 plots the time path of this share (in gray) and its complement (in blue) for each marketplace, level-shifted to zero in January 2025. In all three marketplaces the share of small-audience releases trends upward through 2025: from about 79% to 86% on iOS, from 18% to 31% on Chrome, and by a smaller amount on Android. The matching channel is hence also ruled out. If the additional applications were better-matched substitutes, users switching to them would generate at least the modest demand needed to clear the threshold. Instead, the marginal applications are increasingly invisible to end users.

Taken together, the evidence rules out the principal channels through which the additional releases could have raised consumer surplus. The aggregate-flatness finding rules out both new blockbuster apps and a long-tail effect, and the rising small-audience share rules out a substitution-driven matching gain. The marginal new applications are not, on the whole, reaching end users in meaningful numbers.

---

[21] 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.

8.2.4 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 4: the number of apps across different marketplaces has on average increased over time, but not by as much as the task-level productivity estimates would imply. These increases nevertheless imply that the rise in coding activity we document on GitHub has reached marketplaces.

The second pattern, flat usage, however, admits multiple explanations. The most direct reading is supply-side. AI tools appear to relax upstream constraints in producing and publishing software, raising the number of applications that clear the release threshold. But publication need not be the binding layer 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. In this sense, the marketplace evidence is the aggregate analogue of the attenuation pattern documented above: productivity gains at upstream layers can raise the flow of releases without a proportional increase in adoption-weighted output.[22]

Demand-side congestion, however, is a possible alternative explanation that we cannot rule out. Even if the quality of new applications 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 applications compete for a fixed pool of attention, average per-app usage falls. This mechanism is not a production bottleneck, but it yields the same reduced-form implication: more publications need not translate into proportionally more usage.

Several caveats apply. First, our sample runs only through early 2026, leaving barely a year 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 applications 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 application generates for its users, particularly for free applications.

---

[22] A related supply-side reading is selection on the entry margin. By lowering the fixed cost of producing a minimally publishable application, AI tools may draw in projects with lower expected market value or less capacity for post-release iteration; the new entrants then concentrate in the low-usage tail.

9 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. Yet these gains attenuate sharply across the production hierarchy: sync agents lead to a 741% increase in lines of code and a 65% increase in pull requests, but releases rise by only 20%. We find similar increases in aggregate coding activity on GitHub.

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 0.25, pointing to strong complementarity between AI and human inputs along the production chain.

The same logic motivates our analysis of application marketplaces: if AI increases the supply of software, does that software reach users, and is it actually used? Across four major marketplaces, new application creation has accelerated since mid-2025, but total usage within the first three months has not risen, and the share of new applications that fail to reach even a modest audience has increased. This pattern admits two interpretations: either the marginal applications are of lower quality, or a consumer-side bottleneck—discovery and adoption—prevents increased supply from translating into usage.

Taken together, our results empirically document 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. In software, the binding constraint appears to be shifting from writing code to reviewing, integrating, and ultimately 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 P. Restrepo (2019). Automation and New Tasks: How Technology Displaces and Reinstates Labor. *Journal of Economic Perspectives 33*(2), 3–30.

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.

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

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*.

Bick, A., A. Blandin, and D. J. Deming (2024). The Rapid Adoption of Generative AI. *NBER Working Paper, No. 32966*.

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, 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 (2023). AI Assistance in Legal Analysis: An Empirical Study. *Journal of Legal Education*.

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.

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*. Articles in Advance.

Demirer, M. (2025). Production Function Estimation with Factor-Augmenting Technology: An Application to Markups. *Working Paper*.

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.

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

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

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. *arXiv Preprint arXiv:2511.04427*.

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*.

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). A.I. and Our Economic Future. *NBER Working Paper, No. 34779*.

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

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.

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)*.

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

OpenAI (2025). The State of Enterprise AI. *OpenAI Report*.

Otis, N., R. Clarke, S. Delecourt, D. Holtz, and R. Koning (2025). The Uneven Impact of Generative AI on Entrepreneurial Performance. *SSRN Working Paper, No. 4671369*.

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.

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*.

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

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

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

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 4. 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 (8). The pass-through depends only on the output elasticity θ and the elasticity of substitution σ; the structural share parameter α enters only through θ.

A.3 Proof of Proposition 1

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

Proof of Proposition 1. (a) $\partial G/\partial\theta > 0$.

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

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

(b) $\partial G/\partial\sigma > 0$. The pass-through $G(A, \theta, \sigma) = [\theta A^{\rho} + (1-\theta)]^{1/\rho}$ 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 $\partial G/\partial\sigma > 0$ for all $A > 1$ and $\theta \in (0, 1)$. □

A.4 Proof of Proposition 2

Proof of Proposition 2(a). Consider 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 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 $A \longrightarrow \infty$ in the pass-through formula (14):

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

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

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

□

A.5 Connection to Jones (2026)

The bound (15) generalizes the $\frac{1}{1-s}$ result in Jones (2026). 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 the spending share s. 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 Jones's benchmark ($\sigma = 0.5$), the maximum gain is a doubling. The bound increases sharply as σ approaches the Cobb–Douglas knife-edge ($\sigma = 1$), where it disappears entirely.

A.6 Calibration of (θ, σ)

We calibrate the structural parameters (θ, σ) to match the empirical attenuation pattern observed for autocomplete. Autocomplete enters the production hierarchy only at layer 1 (code writing), so we can take its empirical lines-of-code effect as the layer-1 shock A and use the model to derive predicted gains at every higher layer. For tractability we assume θ and σ are constant across layers, leaving two free parameters.

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):

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

with $G_1 \equiv A$. We then choose (θ,σ) to minimize the sum of squared log differences between predicted and empirical gross gains across the downstream layers:

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

We then use the calibrated parameters to predict the attenuation pattern for sync and async agents, taking each tool's empirical lines-of-code effect as A and applying the same recursion. This exercise asks a counterfactual question: *if* sync and async agents operated only at layer 1, like autocomplete, what attenuation pattern would the model predict at higher layers? Comparing this prediction to the empirical attenuation reveals at which layers each tool intervenes beyond simple pass-through (Figure 10).

B Data Appendix

This appendix describes the data underlying our empirical analysis. We first detail the procedures used to identify users of each of the seven coding tools we study and to collect their GitHub activity. We then describe how we construct the user-week panels used in the event-study analysis and define the outcome variables measured at the developer-week level.

B.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. Constructing the sample involves two steps. We first construct the set of treated users for each AI tool from the tool's adoption data, and we separately construct a pool of candidate control users. We then construct each user's coding activity over the relevant sample period and aggregate it into developer-week panels measuring the outcome variables of interest. The remainder of this appendix proceeds in four 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. Third, we describe how we identify the activity of async coding agents. Finally, we describe the construction of the user-week panels used in our analysis.

B.2 Coding Tool Usage Data

We categorize the seven coding tools into two groups based on their data sources and user identification strategies.

The first group, AI Coding Agents with Direct Usage Signals, includes GitHub Async Agent, Codex Async Agent, Claude Code Sync Agent, and Dockerfile. These tools leave identifiable traces in open-source repositories—through dedicated bot usernames, branch-name prefixes, or specific configuration files—which allow us to identify users directly from public GitHub data.

The second group, AI Coding Tools with Subscription Data, includes GitHub Autocomplete, GitHub Pro, and GitHub Sync Agent. For these tools, we obtain user subscription records from Microsoft, which provide subscription dates but not direct usage signals. For this group, we use the subscription date as the adoption date of the tool.

The remainder of this subsection provides the data collection details for each tool.

B.2.1 GitHub Async Agent

Table OA-1: User-Source Distribution for Copilot Agent PRs and Identified Users

	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 Async Agent (Copilot Agent) pull requests. We apply a priority hierarchy that searches for co-authors first, then requested reviewers, and finally assignees (Appendix B). The PR-level columns count the 694,407 Copilot 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.

GitHub Async Agent (also known as GitHub Copilot Async 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[23].

To identify users, we use the GitHub REST API issues endpoints to retrieve all pull requests created by the GitHub Async 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 PR opened by the Copilot Coding 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 <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.

---

[23] GitHub Docs—About Coding Agent

We exclude the Copilot bot account itself from the user list. Each pull request created by the GitHub Async Agent is linked to one or more human users according to the hierarchy above. 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 Copilot Coding Agent and 2,084,472 commits associated with these pull requests. Applying the priority hierarchy above, we identify 109,746 distinct users. Table OA-1 reports the user-source distribution, and Figure 4(c) shows the distribution of event dates.

B.2.2 Codex Async Agent

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. We exclude pull requests with creation dates outside the feasible observation window—either before 2025-05-16 (the release date of Codex[24]) or after the data collection date.

As of 2025-12-22, we retrieved 2,643,226 pull requests created by 142,902 Codex users. Figure 4(b) shows the distribution of event dates.

B.2.3 Claude Code Sync Agent

When starting a conversation, Claude Code automatically pulls `CLAUDE.md` files into its context[25]. 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.[26].

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 exclude users whose event dates fall before 2025-02-24 (the release of Claude Code) or after the data collection date.

---

[24] 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

[25] Anthropic—Claude Code Best Practices

[26] 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 (190,801,306), 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

We identify 59,614 `CLAUDE.md` files across 51,865 repositories, along with 179,879 associated commits authored by 28,558 unique authors. Figure 4(a) shows the distribution of event dates.

B.2.4 GitHub Sync Agent

We obtain a universe of individual subscribers to GitHub Copilot Sync Agent from Microsoft, comprising 74,296 users with subscription dates spanning 2025-05-15 to 2025-11-30.

B.2.5 GitHub Autocomplete

We obtain a universe of individual subscribers to GitHub Copilot Autocomplete from Microsoft, comprising 244,895 users with subscription dates spanning 2022-06-21 to 2022-12-26.

B.2.6 GitHub Pro

We obtain a universe of individual subscribers to GitHub Pro from Microsoft, comprising 22,007 users with subscription dates spanning 2022-01-04 to 2022-12-26.

B.2.7 Dockerfile

As a placebo test, we study developers who commit to Dockerfile files, which represent a form of code development activity unrelated to AI coding assistance. 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.

We identify 1,942,752 dockerfile files across 1,073,370 repositories, with 5,257,726 associated commits authored by 540,710 unique users.

B.3 Construction of the Control Pool

We draw matched controls for each treated user from a broad universe of GitHub developers, which we construct from the near-complete set of GitHub accounts together with a coarse screen on activity. This universe is also the population within which we search for `CLAUDE.md` files (Claude) and Dockerfiles, as described above.

We first enumerate GitHub accounts. On 2024-12-06, we queried the GitHub REST API user endpoint (`https://api.github.com/users?since=1&per_page=100`) to retrieve every account up to the highest user `id` assigned as of that date, 190,801,306 accounts in total, obtaining the current `login` for 174,130,663 of them (91.26%); the remainder reflect deleted, suspended, or otherwise inaccessible accounts. Because GitHub logins can change over time while numeric `id`s are permanent, we retain this `id`-to-`login` mapping and use it to refresh login-based request keys by `id` throughout data collection.

Collecting commit histories for all 174 million accounts is infeasible under GitHub API rate limits, so we instead screen the account space using GHArchive (GHArchive), a complete public log of GitHub events that can be downloaded in bulk. From GHArchive events spanning 2018 through 2024, we build an actor-level panel recording the months in which each `id` appears as the actor of any event, we keep accounts whose first appearance precedes 2020-01 and whose last appearance follows 2022-12, and we additionally exclude the 12,769 screened accounts for which no `login` was retrieved above. We impose this multi-year span requirement to focus on established, regularly active developers—rather than transient, one-off, or automated accounts—and to ensure that each candidate control is observed both well before and well after the treatment windows, so that it can support pre-trend matching (the median such user records 112 GHArchive events across 2018–2024). The resulting control pool comprises 5,499,509 developers, or 2.88% of all GitHub accounts.

B.4 Outcome Variables

This section describes how we construct the outcome variables used in our analysis for both treated and control developers. We go through each variable in turn and explain its data source.

B.4.1 User-level Commit Data

We use the GitHub REST API List commits endpoint and filter requests by the author query parameter using each agent user's GitHub login to retrieve commit data for agent users. We exclude users with more than 100,000 commits in the observation year or more than 1,000 commits on any single day, as complete retrieval is infeasible under GitHub REST API pagination and rate limits.

Forking a repository copies its commit history verbatim, so a single commit—identical in every field, including its `sha`—can appear 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. We parse the `sha` and the `{owner}/{repo}` repository from each commit `url` and, pooling all retrieved commits, enumerate the distinct repositories in which each `sha` appears. We separately query the GitHub REST API repository endpoint (`/repos/{owner}/{repo}`) to obtain the creation date of every repository appearing in the commit data, and use these dates to select a canonical repository for each `sha` as follows: if the `sha` appears in a single repository, that repository is canonical; if it appears in several whose creation dates are known, we keep the one created earliest—the original, since any fork is created after the repository it copies; if some candidates' creation dates are missing, we keep the earliest among those that are known; and if no creation date is available for any candidate, we keep the alphabetically first repository name. We retain a commit record only if its repository equals the canonical repository for its `sha`. This `sha`-to-repository assignment is constructed once, globally, over the union of all retrieved commits, so that the canonical choice is identical across developers and across all commit-based outcomes. Because the discarded copies are identical to the retained record in author, timestamp, message, line statistics, and file list, this step removes only mechanical fork duplication and leaves genuine activity unchanged. In our data, 18.7% of distinct commits appear in more than one repository and are collapsed to a single canonical copy in this way.

B.4.2 Lines of Code

We measure the total number of lines of code added, changed, or removed by each developer in each week. While this is a crude measure of productivity, it is important in particular to verify that adoption of AI coding tools does not merely lead to more frequent use of version control software (e.g., through more frequent commits in reaction to worries about losing work).

For each commit we follow its `url` (of the form `.../repos/{owner}/{repo}/commits/{sha}`) to the commit-info record, which reports the commit's line statistics as `stats_total` (total lines changed, i.e. additions plus deletions), `stats_additions`, and `stats_deletions`. After the SHA-level deduplication above, we sum `stats_total` across a developer's commits within each week.

B.4.3 Distinct Files

We measure the number of distinct files touched by each developer in each week. This is a crude measure of the scope of a developer's work, and in particular may measure whether a developer (or coding tool) must work on isolated changes or can effectively integrate changes spanning multiple abstractions.

The commit-info record also enumerates the files each commit changes; this file-level data stores, per file, its `filename` together with `status`, `additions`, `deletions`, and `changes`. We identify a file by the pair (repository, `filename`) and count the distinct files a developer touches within each week.

B.4.4 *Commits*

We measure weekly commit counts at the developer level. For users of GitHub 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. We use this decomposition to separately examine changes in human coding activity and the volume of AI-generated code.

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.

B.4.5 *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. 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.

We retrieve pull requests with, for each pull request, 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 by `pull_request_url`, 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.

B.4.6 *Distinct Repositories*

We measure the number of distinct repositories touched by each developer in each week. This allows us to get a sense to what extent 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.

B.4.7 *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—the week itself together with the eleven weeks before it—and count how many distinct repositories in that set had at least one release created during that week. This measure therefore captures how many of the projects a developer has worked on over the trailing twelve weeks shipped a release in the current week.

B.5 Construction of the Joint Panel

This section describes how we assemble the developer-week panel used in our analysis. For each treated and control developer, the panel records four families of weekly activity, each drawn from a distinct collection: commit counts (in total and, for asynchronous-agent users, split into human- and agent-attributed commits); commit-level effort statistics, namely the number of lines changed, distinct files, and distinct repositories; the number of pull requests created; and the number of repositories shipping releases. Throughout, we work on a common weekly grid of 324 consecutive Monday-to-Sunday weeks running from 2020-01-06 through 2026-03-16, assigning each commit, pull request, and release to the week containing its date and dropping events dated outside this range. We then identify the asynchronous coding agents' activity and separate it from human activity, and finally aggregate every outcome to the developer-week level and merge the four families into a single panel restricted to the matched developers. Combined with each treated developer's adoption date as the event time, this panel forms the basis for our event-study estimation.

B.5.1 Identifying and attributing agent activity

We observe the two asynchronous coding agents from their public traces. For the GitHub agent we retrieve every pull request opened by its bot account, `copilot-swe-agent[bot]`, and then the commits contained in each such pull request; Codex pull requests are identified by the `codex/` prefix of their head branch. A pull request and its commits count as agent activity only if dated on or after the agent's public-release date—2025-05-19 for the GitHub agent and 2025-05-16 for Codex—and matches before these dates are not counted as agent activity. Claude leaves no pull-request trace—its adoption is detected from `CLAUDE.md` files—so for Claude we record commits only and perform no agent–human split.

Because the GitHub agent commits under its own bot account rather than as the developer, we reassign its commits to the human who initiated the work. From each bot commit message we parse the `Co-authored-by` trailers, each of which gives a collaborating developer's GitHub numeric identifier, and we exclude the agent's own two bot accounts (numeric identifiers 198982749 and 175728472); a bot commit naming several co-authors is expanded into one record per named developer. We then drop the agent's automated planning commits, whose message is exactly "Initial plan" or "Initial plan for issue." A bot commit that still carries no co-author is assigned the developer named by the other commits in the same pull request when those name exactly one developer, and is dropped when they name more than one or none. Codex commits require no reassignment, as they are recorded under the developer's own account.

With agent commits identified and attributed, we form three commit counts for each developer-week: the total number of commits, the number occurring in GitHub-agent pull requests, and the number occurring in Codex pull requests, where a commit is assigned to an agent when its hash matches that agent's pull-request commits and its date is on or after the agent's release date. The GitHub-agent count includes the bot commits reassigned to the developer, and the developer's commits outside any agent pull request are recovered by subtracting the two agent counts from the total. The commit-effort outcomes—lines changed, distinct files, and distinct repositories—are computed three times in parallel: once on the developer's commits outside agent pull requests, once on the commits in their GitHub-agent pull requests, and once on the commits in their Codex pull requests, so that agent-generated code is measured separately and never enters the developer's own totals.

B.5.2 Aggregation and merging

We aggregate each of these measures to the developer-week level, restrict to the matched set of treated and control developers, and merge the four families on developer and week. We expand the merged data to a rectangular grid over all matched developers and all 324 weeks, so that a developer-week with no record in some family appears as missing rather than dropped. We then trim the panel at the end of data collection, dropping weeks after the last complete week (the week beginning 2026-03-16), apply the analogous end-of-collection trim to the earlier adoption cohort, and for the dockerfile placebo drop the first half of 2024, retaining the second half as its pre-period. Where the same developer-week appears in both the treated and the control commit counts, we keep the larger value.

B.6 Marketplace App 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 two things: the number of apps available over time, and an app-level measure of usage. The specific usage measure differs across stores because each store reports usage in its own way. 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. Counting the number of new apps over time requires the full sequence of historical snapshots: because apps are periodically removed from the store, a single current snapshot's creation-date field would systematically underestimate the rate of new entry.

Every panel is restricted to the common observation window 2020-01 through 2026-05 and uses a consistent monthly aggregation scheme: each app contributes one row per calendar month it is observed, with all attributes summarized to that month. 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, covering the data source, collection method, observed variables, panel construction, sample size, and known limitations.

B.6.1 Google Play Store (Android)

We license weekly snapshots of the Google Play Store from SimilarWeb (previously known as 42matters), a data provider that crawls public store pages on a fixed schedule and returns the resulting catalog as a flat record per app per snapshot.[27] The data we obtain begin on 2020-02-29 and extend through 2026-05-16, comprising 327 weekly snapshots. Each snapshot is intended to be a near-complete crawl of every app available worldwide; an app is recorded in a snapshot only if it was reachable through SimilarWeb's enumeration of the store on that crawl date.

For each app–snapshot record we observe the app's Play Store package name (which we use as the persistent identifier), the developer account that publishes it, its assigned

---

[27] See similarweb.com.

category and the broader type distinguishing Applications from Games, an installation-count range reported by Google in buckets such as "1,000–5,000", "5,000–10,000", "10,000–50,000", and so on up to "1,000,000,000–5,000,000,000" (which we store as a lower and upper bound), a running version count and an indicator for whether the app was updated since the prior snapshot, and rating signals consisting of the average user rating (on the 1–5 scale) and the total number of ratings the app has received. Because Google reports installs only as bounded ranges, the lower bound is a conservative measure of cumulative installs and the upper bound is the corresponding ceiling; for most cohort-level summaries we use the lower bound.

From the weekly snapshots we first build a per-app first/last-seen file by recording, for each package name, the earliest and latest snapshot date in which it appears as its entry and exit dates. This file constitutes the universe of apps observed in the Play Store over the sample window. For the apps in this universe we then construct a monthly panel by retaining, for each app and each calendar month it is observed, the snapshot record from the last weekly crawl of that month (so that monthly variables are pegged to a consistent within-month observation). When an app appears and disappears from the store within the window between its entry and exit dates, attributes for those interim months are back-filled from the most recent prior snapshot. The resulting monthly panel contains 245,375,644 app–month rows covering 9,610,667 unique Android apps over 76 months.

Known limitations include the bounded reporting of installs, which means we cannot distinguish among apps within a Google-reported range; the fact that weekly snapshots may miss apps that are temporarily delisted and re-listed within a single week; and that we observe only the worldwide catalog, not country-level availability.

B.6.2 Apple App Store (iOS)

We license weekly snapshots of the Apple App Store from the same vendor (SimilarWeb), spanning 2020-03-01 through 2026-05-17 across 325 weekly crawls. The collection methodology mirrors that of the Android data: each snapshot is intended to be a near-complete crawl of the worldwide catalog and returns one record per app per snapshot.

For each app–snapshot record we observe the App Store track identifier (used as the persistent id), the developer account that publishes the app, its primary genre and category, the content-advisory rating, and a range of product attributes: the current price, an indicator for whether the app offers in-app purchases, the binary's reported size in bytes, and indicators for iPhone and iPad compatibility. The raw data list every country store in which the app is available; we retain the app's primary country, the total number of countries, and an indicator for whether the U.S. store is among them. Also available are the number of supported languages, the number of supported devices, and the number of distinct in-app features. Rating signals consist of the lifetime average rating and total rating count, the corresponding figures for the current version only, and the star-level breakdown of the rating distribution from one to five stars. We record a per-snapshot update indicator and a running version count. Unlike Google, Apple does not publicly disclose download or install counts, so the iOS panel contains no download variable.

We construct the monthly panel using the same procedure as for Android: per-id first/last-seen dates from the sequence of weekly snapshots, one row per app per observed month using the last weekly crawl of that month, and back-fill from the most recent prior snapshot for any interim months in which the app drops out of the store. The resulting panel contains 150,348,066 app–month rows covering 5,023,025 unique iOS apps. Limitations parallel those on Android: weekly cadence may miss short-lived listings, and the absence of installs means that adoption is measured through rating activity rather than direct user counts.

B.6.3 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.[28] We use two of its feeds. The first, *ranking-stats*, is a daily snapshot of every item available in the store on the crawl date, with the item's full catalog metadata—publisher, category, version, ratings, current user count, and the item-type label—attached to each record. The second, *all-versions*, is a daily snapshot of every item's version-release history: for each item it lists every version ever published with the date that version was released. The all-versions feed carries no metadata beyond name, author, size, and description, but its release dates 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.

We combine the two feeds to construct, for each item, an entry date and an exit date. The entry date is the earlier of the first release date in all-versions and the first appearance in ranking-stats. The exit date is the last appearance in ranking-stats, or, if the item never appears in ranking-stats during our coverage window, the most recent release date observed 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.

For each item–snapshot record from the ranking-stats feed we observe the Chrome

---

[28] See chrome-stats.com.

Web Store id (used as the persistent identifier), an item-type label classifying each id as an extension, an app, or a theme, a finer-grained category assignment (e.g., Productivity, Developer Tools), the developer account that publishes it, the current user count reported by the store, the average user rating and rating count, a running version count, and an update indicator.

The Chrome Web Store distinguishes three item types: extensions, apps, and themes. Apps have been deprecated by Google and no longer receive updated values; themes are visual customizations rather than software. We therefore restrict our analysis to extensions.

The user count requires particular attention because of how the Chrome Web Store reports it. 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 of cumulative installs comparable to the other marketplaces, we take the running maximum of the per-snapshot user count for each extension over the observation window and treat that running maximum as cumulative installs.

Panel construction follows the same template as the Android and iOS datasets, applied to the combined entry/exit dates above: we build a monthly panel with one row per extension per observed month (pegged to the last ranking-stats snapshot of that month), and back-fill any interim months in which the extension drops out of ranking-stats from the most recent prior snapshot.

Two limitations are specific to the Chrome data. First, although the user count is reported as an integer, it is only exact for small extensions; for larger ones the value is rounded to bucket boundaries (10,000, 20,000, . . . , 100,000, 200,000, . . .), so two extensions whose user counts fall in the same bucket are indistinguishable. Second, ranking-stats coverage begins on 2023-03-31, so all attribute data are available only from that date onward; entry dates extend earlier through the all-versions feed but those earlier months carry no metadata.

B.6.4 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 330,447 distinct projects. For each project we then fetch the monthly download statistics from SourceForge's public JSON statistics endpoint over the window 2020-01-01 through 2026-05-31, scrape the project's HTML overview page for its category and tag assignments, summary description, listed license, registration date, and publicly visible maintainer information, and 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), a monthly time series of downloads spanning the 77 months from 2020-01 through 2026-05, a fixed set of metadata attributes (category, tags, license, description, registration date), and an event history of file releases (one row per release with date, tag, and filename).

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

C Estimation Details

C.1 Overview

This appendix describes the estimation procedure underlying the matched difference-in-differences event studies in the main text. We proceed in four steps. Section C.2 explains how we identify treated developers, construct the donor pool, and assemble matched treated-control pairs. Section C.3 defines the estimation sample, including tool-specific calendar windows, the active-span and zero-padding rule, and the end-of-sample buffer. The remaining subsections present the event-study, interrupted time series, post-period binned, pooled, and quartile-heterogeneity specifications, and record the inference and winsorization conventions used throughout.

C.2 Matched Control Groups

C.2.1 Two matching strategies

The matching approach depends on how treatment is identified in the underlying data. Where treatment timing comes from *direct observation of tool use* (Claude Code, Codex, GitHub Async Agent, Dockerfile), we know the calendar date of the first tool-assisted submission and exact-day matching is feasible. Where treatment is identified from *subscription records* (GitHub Autocomplete, GitHub Pro, GitHub Sync Agent), the subscription date may not coincide with any coding activity in public repositories, so we match at the week level and add an activity requirement at the treatment week (and the corresponding week one year earlier for matched controls):

Strategy A – exact same-day PR matching. Treated developers are matched to controls who authored a pull request on the *same calendar date* one year earlier. No additional activity requirement is imposed on the treated developer beyond the event definition.

Strategy B – same-week commit-and-PR activity matching. Treated and control developers are required to have both at least one commit and at least one pull request during the (pseudo-)treatment week. This ensures matched controls were actively engaged in substantial coding work in the week that corresponds to the treated developer's subscription start.

In both strategies, controls are drawn from the calendar year preceding the treated period. This one-year shift ensures that seasonal patterns in developer activity (holiday effects, academic calendars, quarterly business cycles) are balanced between treatment

Table OA-2: Matching Parameters by Coding Agent

Agent	Strategy	Treated	Control	Match Level	Activity Req.
Claude Code	A	2025	2024	Exact date	PR only
Codex	A	2025	2024	Exact date	PR only
GitHub Async Agent	A	2025	2024	Exact date	PR only
Dockerfile	A	2025	2024	Exact date	PR only
GitHub Autocomplete	B	2022	2021	Week	Commit + PR
GitHub Pro	B	2022	2021	Week	Commit + PR
GitHub Sync Agent	B	2025	2024	Week	Commit + PR

Notes: "Strategy" indicates whether exact same-day PR matching (A) or same-week commit-and-PR activity matching (B) is used. "Match Level" indicates whether matching occurs at the day level (exact calendar date) or week level (calendar week). "Activity Req." refers to the strategy-specific activity required at the (pseudo-)treatment date or week: "PR only" means no restriction on the treated developer beyond the event definition and a pull request on the corresponding date for the control; "Commit + PR" means both a commit and a pull request during that week are required of treated and control developers alike. In addition, all candidate controls must satisfy the pool-eligibility conditions described in Section C.2.2.

and control groups, which is particularly important because treatment dates for some tools span multiple months over which productivity patterns may vary systematically.

Table OA-2 reports the strategy, treated and control calendar years, match level, and activity requirement for each tool.

C.2.2 Control pool eligibility

The donor pool consists of GitHub developers in our user universe who created at least one pull request during the donor period – one year prior to the treated period – and never adopted any of the coding tools under study. Pull request data are assembled via the GitHub REST API.

Beyond the strategy-specific requirements stated in Section C.2.1, eligibility imposes two further conditions on every candidate control j, applied identically across strategies:

1. **Pre-treatment commit floor.** Developer j has at least one week with strictly positive commits at or before week -11 (the Monday 77 days prior to the pseudo-treatment week, using ISO week conventions), so that matches are restricted to developers with an established commit history on the platform rather than brand-new accounts joining at the pseudo-treatment date.

2. **Panel presence.** Developer j has at least one observation – possibly with zero commits – in the commit panel for the pseudo-treatment week, ensuring they were active on GitHub during that week.

C.2.3 Matched-pair algorithm

We pair treated and control developers using a sequential greedy algorithm with randomized ordering:

1. **Randomization.** Randomly shuffle the order of treated developers so that the matching sequence does not favor developers appearing early in the data.
2. **Sequential matching.** For each treated developer i in the shuffled order:
   (a) Identify the treatment date t_i^* (Strategy A) or treatment week w_i^* (Strategy B).
   (b) Retrieve the set of eligible controls $\mathcal{C}_i$ from the pre-computed eligibility table (constructed per Sections C.2.1 and C.2.2).
   (c) Remove any control developer already matched (matching without replacement).
   (d) For each eligible control $j \in \mathcal{C}_i$, compute the absolute difference in pre-treatment commits $\Delta_{ij} = \left|\text{PreCommits}_i - \text{PreCommits}_j\right|$, where

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

   (e) Select the control j^* with the smallest difference,

$$j^* = \arg\min_{j \in \mathcal{C}_i} \Delta_{ij}.$$

   (f) Record the matched pair (i, j^*) with the pseudo-treatment date or week set equal to developer i's treatment date or week shifted by one year.

C.3 Constructing the Estimation Sample

Before estimation, we apply three restrictions to the developer-week panel constructed in Appendix B that together determine which developer-weeks enter the regressions and how weeks without recorded activity are treated. These restrictions are applied identically to treated and matched control developers.

Tool-Specific Active Periods For each tool g, we confine the analysis to a treated calendar window $[\underline{c}_g^T, \overline{c}_g^T]$ and a parallel control window $[\underline{c}_g^C, \overline{c}_g^C]$ shifted one year earlier, reported in Table OA-3. This confines the active-span and pre-period computations below to the tool-relevant era, so that activity from unrelated calendar periods does not enter a developer's outcome series.

Table OA-3: Tool-Specific Active Periods and End-of-Sample Cutoff

Tool	Treated Window	Control Window	Last Adoption $\max_i a_i$	Cutoff c_g
GitHub Autocomplete	2022-01-03 – 2023-03-20	2021-01-04 – 2022-03-21	2022-12-26	2023-02-06
GitHub Pro	2021-06-28 – 2023-03-20	2020-06-22 – 2022-03-21	2022-12-26	2023-02-06
Claude Code Sync Agent	2025-01-06 – 2026-03-16	2024-01-08 – 2025-03-17	2025-12-22	2026-02-02
Codex Async Agent	2025-01-06 – 2026-03-16	2024-01-08 – 2025-03-17	2025-12-22	2026-02-02
GitHub Async Agent	2025-01-06 – 2026-03-16	2024-01-08 – 2025-03-17	2025-12-22	2026-02-02
GitHub Sync Agent	2025-01-06 – 2026-03-16	2024-01-08 – 2025-03-17	2025-11-30	2026-01-12
Dockerfile	2024-07-01 – 2026-03-16	2023-06-26 – 2025-03-17	2025-12-22	2026-02-02

Active Span and Zero-Padding The panel records a developer-week as missing whenever no activity is observed, which does not distinguish a week of genuine inactivity from a week in which the developer was not present on the platform. To separate the two, we define each developer's *active span* as the interval between the first and last week with strictly positive commits, $[\underline{t}_i, \bar{t}_i]$, computed within the tool's calendar window. Inside this span, weeks with no recorded activity are set to zero: the developer is known to be active, so a missing week is a true zero rather than non-observation, and zero-padding prevents the outcome mean from being mechanically inflated by the omission of inactive weeks. Outside the span—before $\underline{t}_i$ or after $\bar{t}_i$—observations are set to missing and excluded, since a zero there would conflate platform absence with zero productivity.

End-of-Sample Buffer Beyond the end-of-collection trim described in Appendix B, we impose an additional buffer at the right edge of the panel. Because adoption is staggered while the panel has a fixed right edge, event-time cells in the final calendar weeks are populated by a non-random, time-varying subset of cohorts—only the earliest adopters reach the longest post-adoption horizons—and the most recent weeks remain subject to incomplete commit records. Both forces induce end-of-sample (compositional) bias in the estimated dynamics. We therefore impose a common calendar cutoff

$$c_g = \text{NextMonday}\left(\max_i a_i + 6 \text{ weeks}\right), \tag{16}$$

where $\max_i a_i$ is the latest adoption date in tool g's matched sample, and cap each developer's last analyzed event-week at $\lfloor (c_g - a_i)/7 \rfloor$, discarding observations beyond it. The regressions thus retain roughly six weeks of post-adoption data following the final event date, while the roughly six additional weeks collected beyond the cutoff—through the end of data collection—are used only to determine each developer's active span (in particular, the last week with positive commits) and are themselves excluded from estimation. This design ensures that the latest-adopting cohort contributes at most six post-adoption weeks and that no event-time cell rests on a materially thinned or incompletely recorded set of developers, while still using all collected data to establish whether each developer remained active through the end of the analysis window. The cutoff is applied to treated and control developers using the treated developer's adoption date.

C.4 Event Studies

For outcomes with matched controls, we estimate a generalized difference-in-differences event study. Let Y_{it} denote the outcome for treated developer i and $Y_{it}^{control}$ denote the outcome for the matched control.

Level Specification The level difference regression is:

$$Y_{it} - Y_{it}^{control} = \sum_{k=-11, k\neq -1}^{30} \beta_k \cdot \mathbf{1}\{t - t_i^* = k\} + \gamma_{pre} \cdot \mathbf{1}\{t < t_i^* - 11\} + \gamma_{post} \cdot \mathbf{1}\{t > t_i^* + 30\} + \alpha_i + \varepsilon_{it} \tag{17}$$

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 base period is $k = -1$. Standard errors are clustered at the developer level.

Normalized Specification To facilitate cross-developer comparison and address heterogeneity in baseline activity, we normalize each arm by its own pre-period mean:

$$\widetilde{Y}_{it} = \frac{Y_{it}}{\bar{Y}_i^{pre}}, \quad \widetilde{Y}_{it}^{control} = \frac{Y_{it}^{control}}{\bar{Y}_i^{pre,control}}, \quad \text{where} \quad \bar{Y}_i^{pre} = \frac{1}{11} \sum_{k=-11}^{-1} Y_{i,t_i^*+k} \tag{18}$$

and $\bar{Y}_i^{pre,control}$ is defined analogously for the matched control.

The normalized event study specification is:

$$\widetilde{Y}_{it} - \widetilde{Y}_{it}^{control} = \sum_{k=-11, k\neq -1}^{30} \beta_k \cdot \mathbf{1}\{t - t_i^* = k\} + \gamma_{pre} \cdot \mathbf{1}\{t < t_i^* - 11\} + \gamma_{post} \cdot \mathbf{1}\{t > t_i^* + 30\} + \alpha_i + \varepsilon_{it} \tag{19}$$

The coefficients β_k in the normalized specification can be interpreted as proportional changes relative to the pre-period baseline: $\beta_k = 0.5$ indicates a 50 percentage point increase relative to pre-period average.

De-meaning Adjustment We de-mean the estimated coefficients by subtracting the average of pre-period coefficients $\{\hat{\beta}_{-11}, \ldots, \hat{\beta}_{-1}\}$ to center the pre-period estimates around zero:

$$\tilde{\beta}_k = \hat{\beta}_k - \frac{1}{11} \sum_{j=-11}^{-1} \hat{\beta}_j \tag{20}$$

C.5 Interrupted Time Series (ITS) Specification

For the async results (commits in agent-authored PRs), we do not have well-defined controls, and the outcome is mechanically zero in the pre-period (since agent-authored PRs only exist post-adoption), we employ an interrupted time series design without differencing:

$$Y_{it} = \sum_{k=-11}^{30} \beta_k \cdot \mathbf{1}\{t - t_i^* = k\} + \varepsilon_{it} \tag{21}$$

Note that this specification is fully saturated, so the coefficients β_k directly represent the average level of Y_{it} at each event week. Standard errors are clustered at the developer level.

We then normalize Y_{it} by the developer's pre-period mean from the matched human-outcomes panel ($\bar{Y}_i^{pre}$ from equation 18), rather than the async outcome's own pre-period mean (which is mechanically zero), and report this normalized version.

C.6 Post-Period Binned Regression

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

$$\begin{aligned}\widetilde{Y}_{it} - \widetilde{Y}_{it}^{control} = \quad & \beta_0 \cdot \mathbf{1}\{t - t_i^* = 0\} \\ & + \beta_{1\text{-}10} \cdot \mathbf{1}\{1 \leq t - t_i^* \leq 10\} \\ & + \beta_{11\text{-}20} \cdot \mathbf{1}\{11 \leq t - t_i^* \leq 20\} \\ & + \beta_{21\text{-}30} \cdot \mathbf{1}\{21 \leq t - t_i^* \leq 30\} \\ & + \gamma_{pre} \cdot \mathbf{1}\{t < t_i^* - 11\} + \gamma_{post} \cdot \mathbf{1}\{t > t_i^* + 30\} + \alpha_i + \varepsilon_{it}\end{aligned} \tag{22}$$

The omitted category is the pre-period (weeks −11 to −1). The coefficient $\beta_{1\text{-}10}$ represents the average treatment effect during weeks 1–10 post-adoption, and similarly for other post-period bins.

For ITS outcomes (no matched controls), the binned specification omits differencing and fixed effects, and normalizes the outcome by the developer's pre-period mean from the matched human-outcomes panel ($\bar{Y}_i^{pre}$ from equation 18):

$$\begin{aligned}\frac{Y_{it}}{\bar{Y}_i^{pre}} = \quad & \beta_0 \cdot \mathbf{1}\{t - t_i^* = 0\} + \beta_{1\text{-}10} \cdot \mathbf{1}\{1 \leq t - t_i^* \leq 10\} \\ & + \beta_{11\text{-}20} \cdot \mathbf{1}\{11 \leq t - t_i^* \leq 20\} + \beta_{21\text{-}30} \cdot \mathbf{1}\{21 \leq t - t_i^* \leq 30\} \\ & + \gamma_{pre} \cdot \mathbf{1}\{t < t_i^* - 11\} + \gamma_{post} \cdot \mathbf{1}\{t > t_i^* + 30\} + \varepsilon_{it} \end{aligned} \tag{23}$$

where the intercept absorbs the pre-period mean and the post-bin coefficients represent deviations from it.

C.7 Pooled Analysis Specification

For pooled analysis across multiple coding agents, we stack observations from different agent-developer pairs and modify the fixed effects structure. Let $j \in \{\text{Claude}, \text{GitHub Agent}, \text{Codex}, \text{GitH}$ index the coding agent. The pooled matched specification is:

$$\widetilde{Y}_{ijt} - \widetilde{Y}_{ijt}^{control} = \sum_{k=-11, k\neq -1}^{30} \beta_k \cdot \mathbf{1}\{t - t_{ij}^* = k\} + \gamma_{pre} \cdot \mathbf{1}\{t < t_{ij}^* - 11\} + \gamma_{post} \cdot \mathbf{1}\{t > t_{ij}^* + 30\} + \alpha_{ij} + \varepsilon_{ijt} \tag{24}$$

where α_{ij} denotes developer-by-agent fixed effects. This allows the same developer to appear multiple times if they adopt multiple coding agents, with each adoption treated as an independent observation. Standard errors are clustered at the developer-agent level. For pooled async outcomes, the ITS and ITS binned specifications (models 21 and 23) are used with standard errors clustered at the developer-by-agent level but without fixed effects.

C.8 Heterogeneity by Pre-Period Activity

To examine how treatment effects vary with baseline developer activity, we estimate a quartile interaction specification. We first assign each developer i to a quartile $q \in \{1, 2, 3, 4\}$ based on their pre-period activity level:

$$\bar{M}_i^{pre} = \frac{1}{2}\left(\bar{Y}_i^{pre} + \bar{Y}_i^{pre,control}\right) \tag{25}$$

where $\bar{Y}_i^{pre}$ and $\bar{Y}_i^{pre,control}$ are the treated and control pre-period means from equation 18. Quartile breakpoints are at the 25th, 50th, and 75th percentiles of $\bar{M}_i^{pre}$. For ITS outcomes, quartile assignment uses the matched human-commits panel as the reference (since the async outcome is mechanically zero in the pre-period).

We then interact quartile indicators with the post-period bins from equation 22. For matched outcomes, the quartile interaction specification is:

$$\begin{aligned}\widetilde{Y}_{it} - \widetilde{Y}_{it}^{control} = \quad & \sum_{q=1}^{4} \sum_{b \in \mathcal{B}} \beta_{q,b} \cdot Q_q(i) \cdot \mathbf{1}\{t - t_i^* \in b\} \\ & + \gamma_{pre} \cdot \mathbf{1}\{t < t_i^* - 11\} + \gamma_{post} \cdot \mathbf{1}\{t > t_i^* + 30\} + \alpha_i + \varepsilon_{it} \qquad (26)\end{aligned}$$

where $Q_q(i) = 1$ if developer i belongs to quartile q, and

$$\mathcal{B} = \big\{\{0\}, \{1, \ldots, 10\}, \{11, \ldots, 20\}, \{21, \ldots, 30\}\big\}$$

indexes the post-period bins. The omitted category is the full pre-period (weeks −11 to −1) for all quartiles. Standard errors are clustered at the developer level.

Since the dependent variable is normalized by the pre-period mean, the coefficients $\beta_{q,b}$ are in units of the pre-period baseline. We report $100 \times \beta_{q,\text{21-30}}$ as the percentage treatment effect for quartile q in weeks 21–30. For ITS outcomes, the specification replaces the differenced normalized outcome with the level outcome (or its external-baseline-normalized variant) and retains developer fixed effects.

C.9 Winsorization and Estimation

We winsorize dependent variables at the 1% level. For matched difference specifications, we winsorize the treated-minus-control difference at both tails (1st and 99th percentiles). For ITS specifications, we winsorize at the upper tail only (99th percentile), computed over post-period positive values. All regressions are estimated using the `fixest` package in R.

D Robustness Checks

In this appendix, we present the full set of robustness checks summarized in Section 6. Our analysis relies on observational data, so these checks are intended to support a causal interpretation of our estimates.

D.1 Activity bias

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 7 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 7(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 7(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 109.1%, the Docker effect stabilizes at

Table OA-4: Productivity Effects Across Production Layers for Other Coding Tools

	Outcomes (Weeks 21–30, %)					
	Lines Chg	Dist Files	Commits	PRs	Dist Repos	Releases
GitHub Pro	179.1 (80.9)	25.8 (15.5)	14.2 (10.9)	−23.4 (15.0)	3.6 (6.0)	7.5 (15.2)
Docker	182.6 (31.2)	67.8 (6.4)	23.3 (3.8)	22.1 (3.8)	3.9 (1.5)	12.4 (4.7)

Notes: This table reports event-study coefficients (1% winsorized, weeks 21–30 bin, normalized by pre-period mean) measuring the effect of adoption of non-AI coding tools, included as placebo comparisons. Lines Chg = weekly total lines changed, Dist Files = distinct files, Commits = weekly commits, PRs = weekly PRs created, Dist Repos = distinct repositories contributed to, Releases = weekly repo release count.

approximately 23.3%. 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 85.8% increase in productivity from the adoption of coding agents.

In Appendix Figures OA-12 and OA-13, we compare level effects rather than normalized effects for AI coding tools and placebo tools, respectively. We find no detectable level effect from either GitHub Pro or Docker adoption.[29] However, we continue to find a large and persistent effect from the adoption of all AI coding tools.

Table OA-4 reports the same outcomes across production layers for GitHub Pro and Docker. While some attenuation is visible, the effects at every layer are much smaller than those of AI coding tools, reinforcing that the large gradient documented in Table 5 is not an artifact of our research design.

Long-term stability of the effect. To the extent that the effects in Table 4 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 five weeks (see Figure 7). 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".

D.2 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. To assess the robustness of this specification, we estimate the same percentage effect in a different way: we first estimate a treatment effect in levels separately for each pre-period-mean decile—allowing the level effect to differ across baseline-activity groups—then normalize

---

[29] Differences between normalized and level plots can arise because level plots place greater weight on highly productive developers, who may experience smaller marginal gains from the adoption of Docker.

each decile-specific estimate by that decile's treated pre-mean, and finally average across deciles. Table OA-5 reports the resulting estimates for the pooled sync effect across the production hierarchy.

The pooled normalized effect (i.e., the specification we have discussed so far) 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. As a side benefit, the decile-specific estimates in Table OA-5 can also be used to compute other averages of interest—for instance, restricting attention to the top decile recovers the average percentage effect among developers who were highly active in the pre-period.

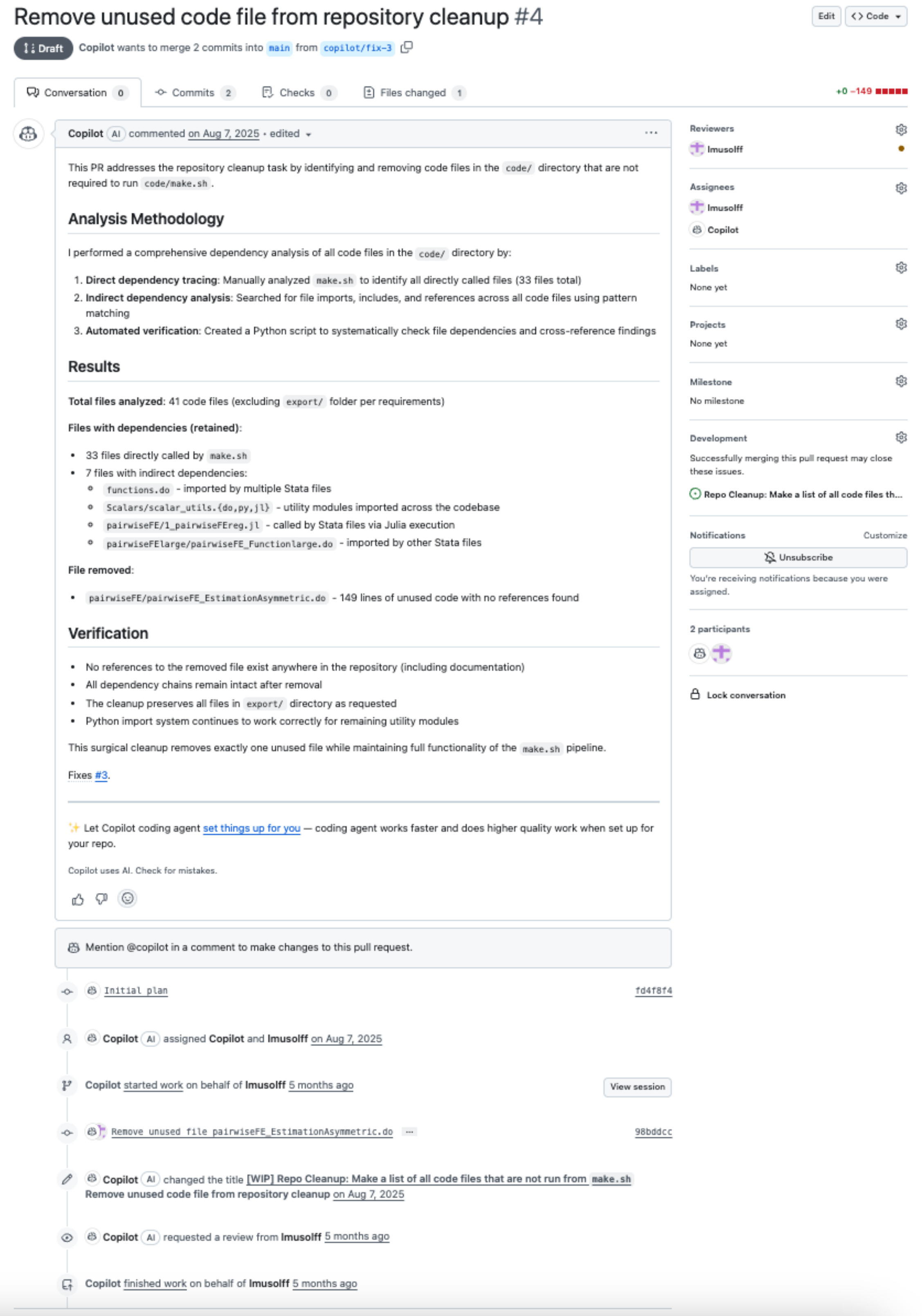
D.3 External validity: public vs. private repositories

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.

E Additional Figures and Tables

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

Figure OA-1: Example of a PR drafted by Copilot



Remove unused code file from repository cleanup #4

Copilot wants to merge 2 commits into `main` from `copilot/fix-3`

Copilot AI commented on Aug 7, 2025 • edited

This PR addresses the repository cleanup task by identifying and removing code files in the `code/` directory that are not required to run `code/make.sh`.

Analysis Methodology

I performed a comprehensive dependency analysis of all code files in the `code/` directory by:

1. **Direct dependency tracing**: Manually analyzed `make.sh` to identify all directly called files (33 files total)
2. **Indirect dependency analysis**: Searched for file imports, includes, and references across all code files using pattern matching
3. **Automated verification**: Created a Python script to systematically check file dependencies and cross-reference findings

Results

Total files analyzed: 41 code files (excluding `export/` folder per requirements)

Files with dependencies (retained):

- 33 files directly called by `make.sh`
- 7 files with indirect dependencies:
    - `functions.do` - imported by multiple Stata files
    - `Scalars/scalar_utils.{do,py,jl}` - utility modules imported across the codebase
    - `pairwiseFE/1_pairwiseFEreg.jl` - called by Stata files via Julia execution
    - `pairwiseFElarge/pairwiseFE_Functionlarge.do` - imported by other Stata files

File removed:

- `pairwiseFE/pairwiseFE_EstimationAsymmetric.do` - 149 lines of unused code with no references found

Verification

- No references to the removed file exist anywhere in the repository (including documentation)
- All dependency chains remain intact after removal
- The cleanup preserves all files in `export/` directory as requested
- Python import system continues to work correctly for remaining utility modules

This surgical cleanup removes exactly one unused file while maintaining full functionality of the `make.sh` pipeline.

Fixes #3.

✨ Let Copilot coding agent set things up for you — coding agent works faster and does higher quality work when set up for your repo.

Copilot uses AI. Check for mistakes.

Mention @copilot in a comment to make changes to this pull request.

`Initial plan` fd4f8f4

Copilot AI assigned **Copilot** and **Imusolff** on Aug 7, 2025

Copilot started work on behalf of **Imusolff** 5 months ago

`Remove unused file pairwiseFE_EstimationAsymmetric.do` … 98bddcc

Copilot AI changed the title [WIP] Repo Cleanup: Make a list of all code files that are not run from `make.sh` **Remove unused code file from repository cleanup** on Aug 7, 2025

Copilot AI requested a review from **Imusolff** 5 months ago

Copilot finished work on behalf of **Imusolff** 5 months ago

Reviewers
Imusolff

Assignees
Imusolff
Copilot

Labels
None yet

Projects
None yet

Milestone
No milestone

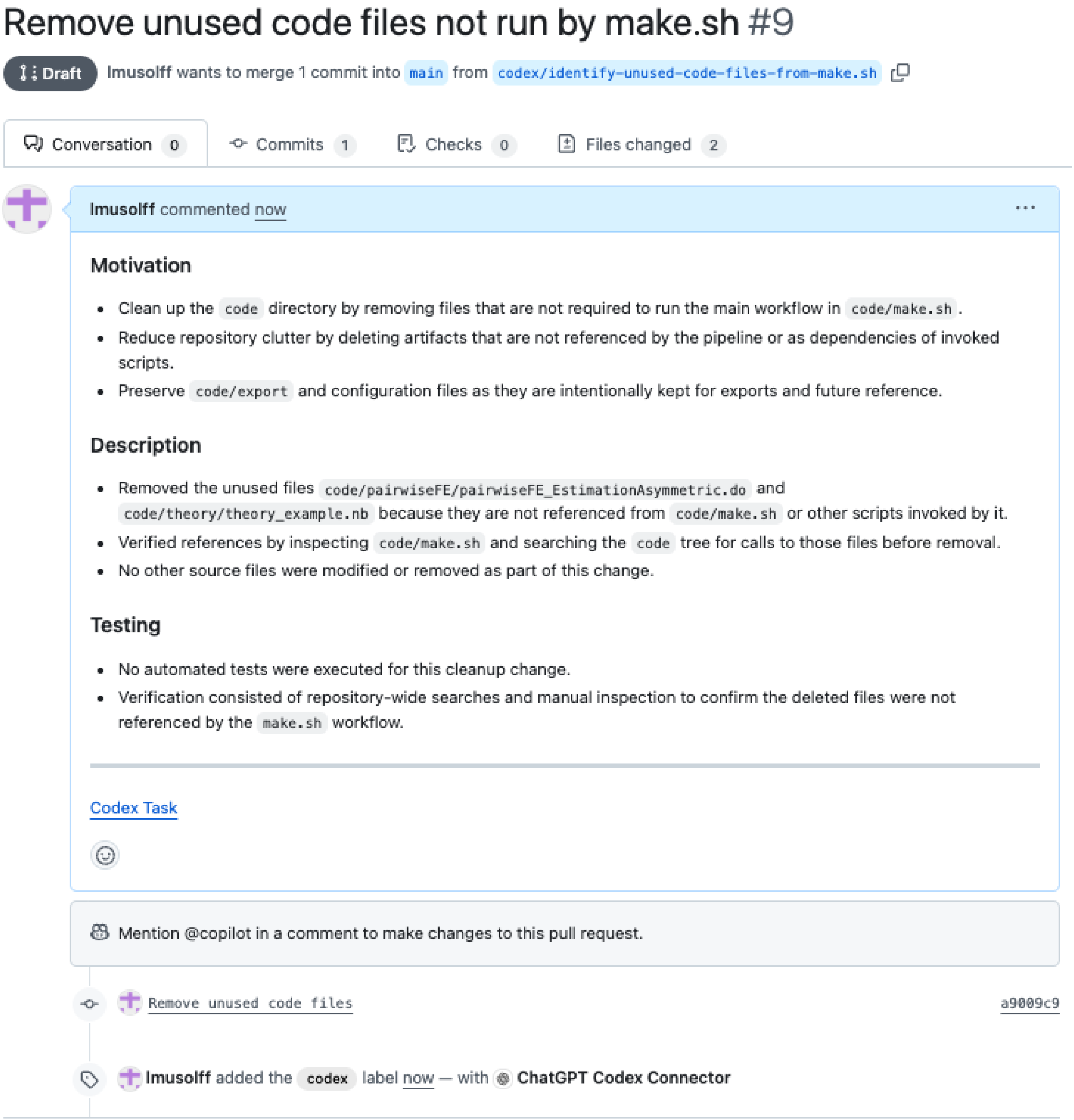
Development
Successfully merging this pull request may close these issues.
Repo Cleanup: Make a list of all code files th...

Notifications
You're receiving notifications because you were assigned.

2 participants

Notes: This figure provides an example of a pull request that has been drafted by GitHub Async Agent.

Figure OA-2: Example of a PR drafted by Codex Async Agent



Remove unused code files not run by make.sh #9

lmusolff wants to merge 1 commit into `main` from `codex/identify-unused-code-files-from-make.sh`

lmusolff commented now

Motivation

- Clean up the `code` directory by removing files that are not required to run the main workflow in `code/make.sh`.
- Reduce repository clutter by deleting artifacts that are not referenced by the pipeline or as dependencies of invoked scripts.
- Preserve `code/export` and configuration files as they are intentionally kept for exports and future reference.

Description

- Removed the unused files `code/pairwiseFE/pairwiseFE_EstimationAsymmetric.do` and `code/theory/theory_example.nb` because they are not referenced from `code/make.sh` or other scripts invoked by it.
- Verified references by inspecting `code/make.sh` and searching the `code` tree for calls to those files before removal.
- No other source files were modified or removed as part of this change.

Testing

- No automated tests were executed for this cleanup change.
- Verification consisted of repository-wide searches and manual inspection to confirm the deleted files were not referenced by the `make.sh` workflow.

Codex Task

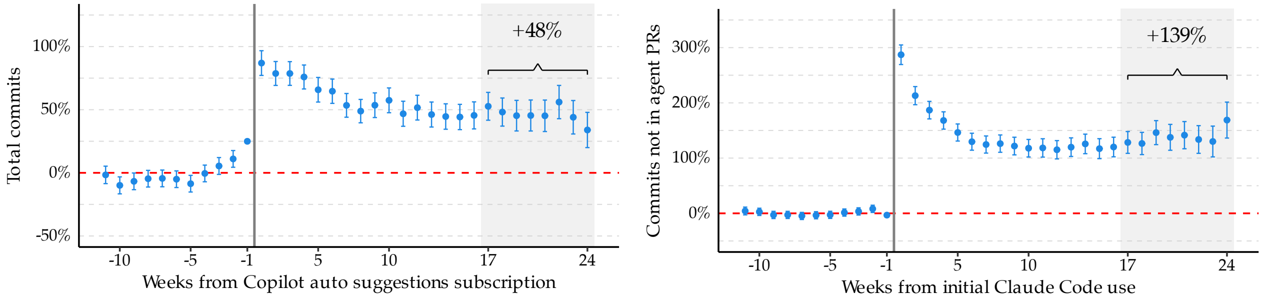
Mention @copilot in a comment to make changes to this pull request.

`Remove unused code files` `a9009c9`

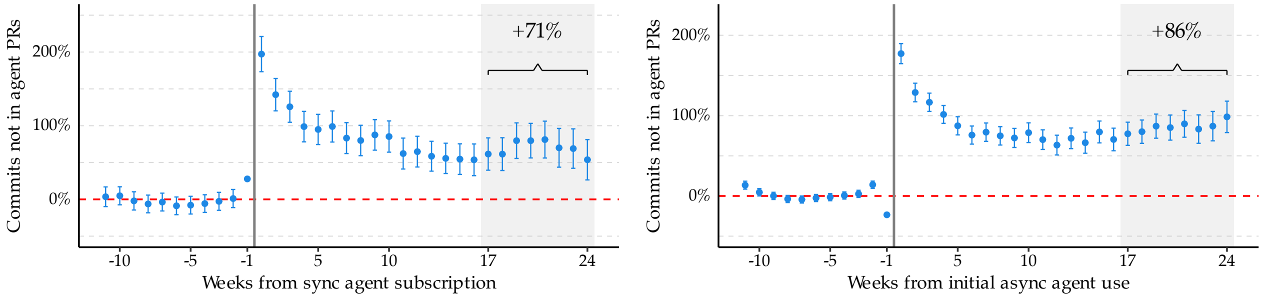
lmusolff added the `codex` label now — with ChatGPT Codex Connector

Notes: This figure provides an example of a pull request that has been drafted by Codex Async Agent.

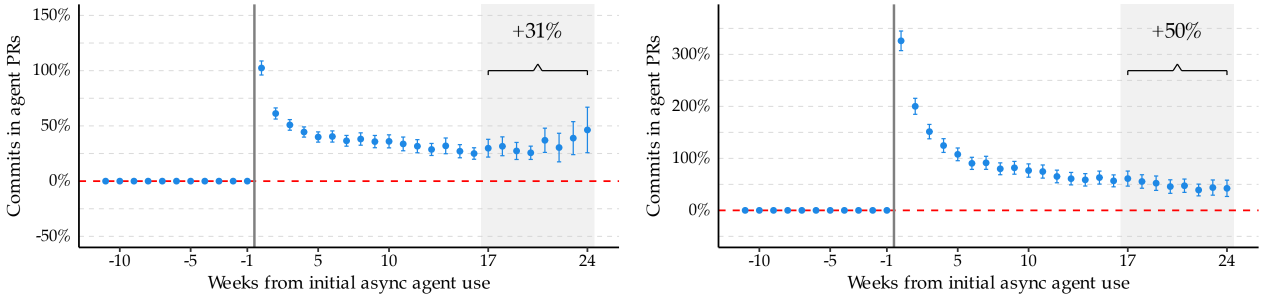
Figure OA-3: Productivity Effects by Tool (Normalized)



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



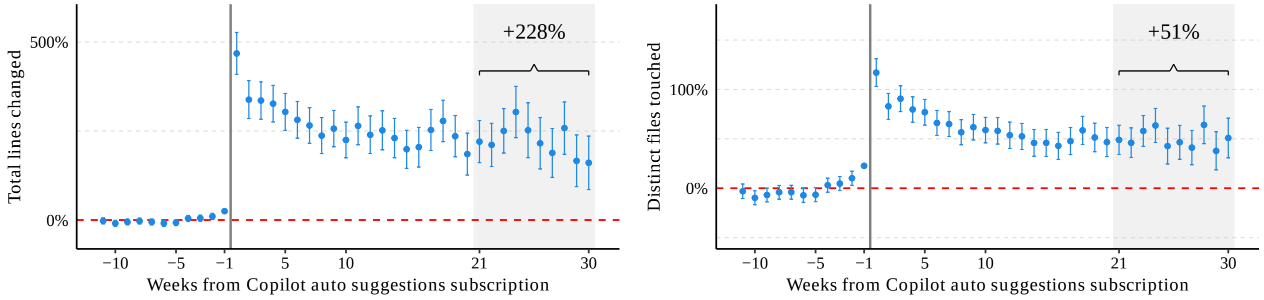
(c) GitHub Sync **(d)** OpenAI Codex (Sync)



(e) GitHub Agent (Async) **(f)** OpenAI Codex (Async)

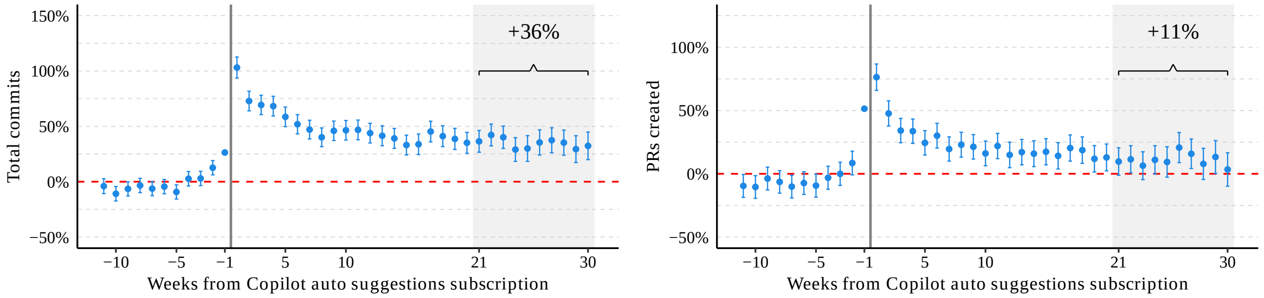
Notes: This figure presents per-tool matched event-study estimates of the effect of adopting each AI coding tool on commits, normalized by each developer's pre-period mean. The top row shows autocomplete and Claude Code; the middle row shows GitHub Sync and OpenAI Codex sync effects (human-authored commits); 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.

Figure OA-4: GitHub Autocomplete Across Production Layers



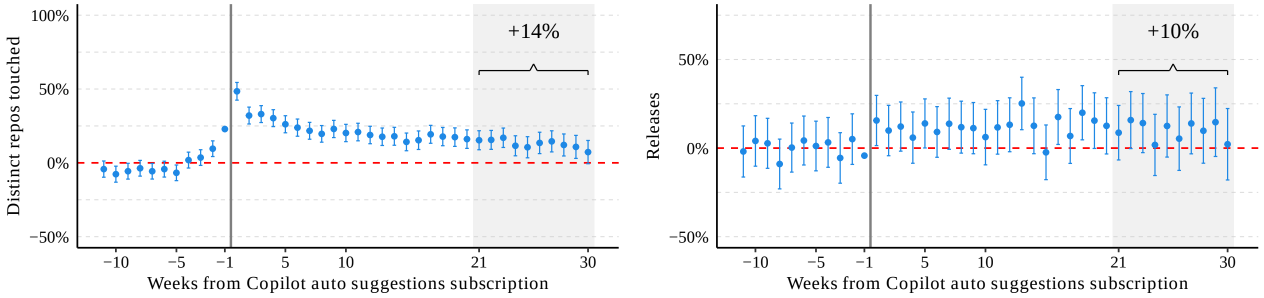
(a) Lines Changed

(b) Distinct Files



(c) Commits

(d) Pull Requests Created

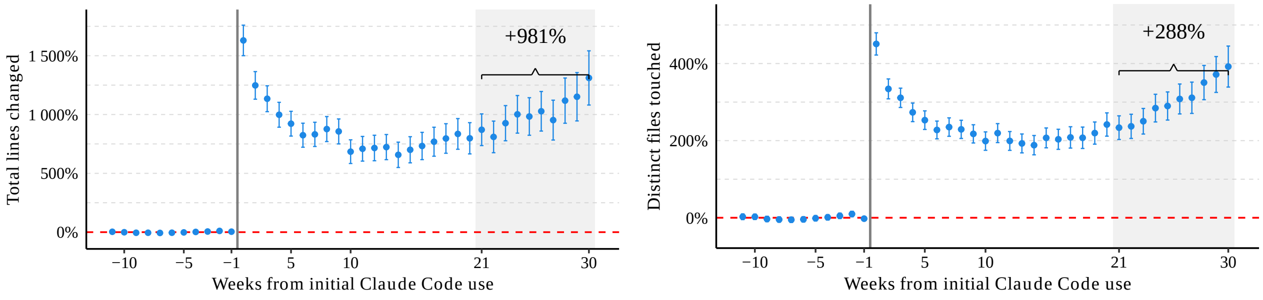


(e) Distinct Repositories

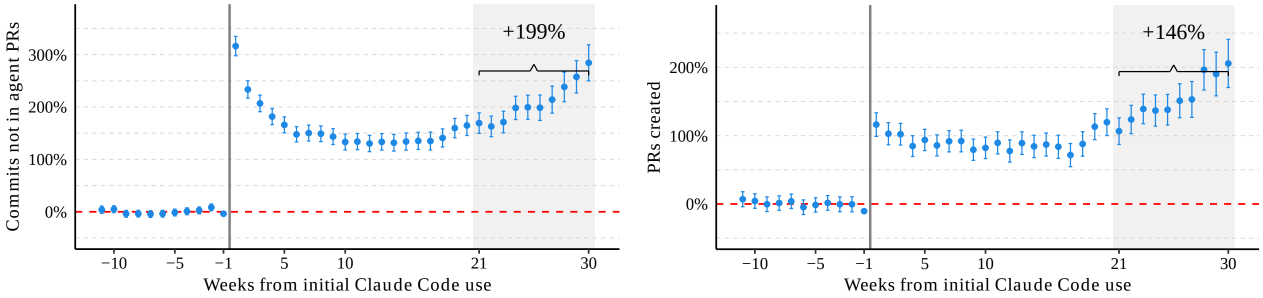
(f) Releases

Notes: This figure presents matched event-study estimates of the effect of GitHub Autocomplete adoption across production layers, replicating Figure 8 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 576.5%, 89.2%, 87.0%, −21.8%, −7.2%, and 8.4% for Panels OA-4(a)–OA-4(f), respectively.

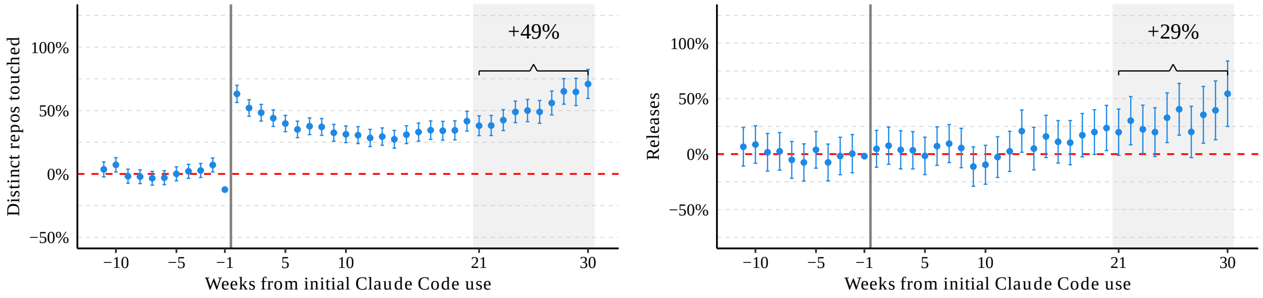
Figure OA-5: Claude Code Across Production Layers



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



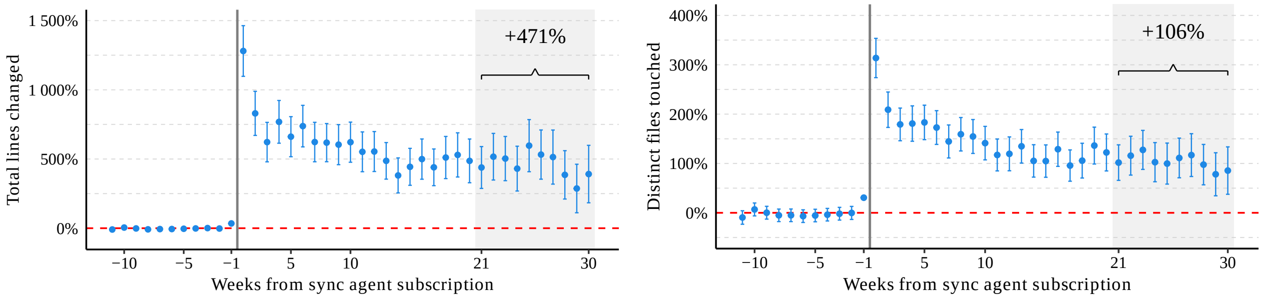
(c) Commits **(d)** Pull Requests Created



(e) Distinct Repositories **(f)** Releases

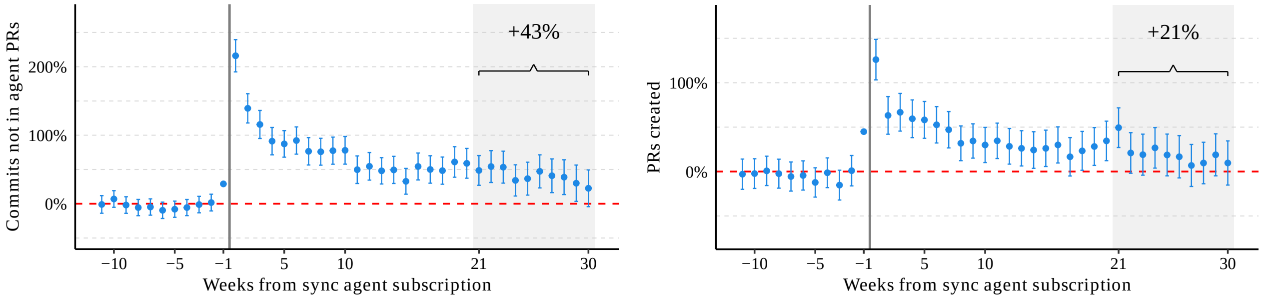
Notes: This figure presents matched event-study estimates of the effect of Claude Code adoption across production layers, replicating Figure 8 for the individual tool. Outcomes are normalized by each developer's pre-period mean. The first post-adoption week is omitted as transitory effects distort the scale; the omitted coefficients are 3,835.9%, 960.4%, 523.6%, −77.2%, 78.5%, and −3.8% for Panels OA-5(a)–OA-5(f), respectively.

Figure OA-6: GitHub Sync Agent Across Production Layers



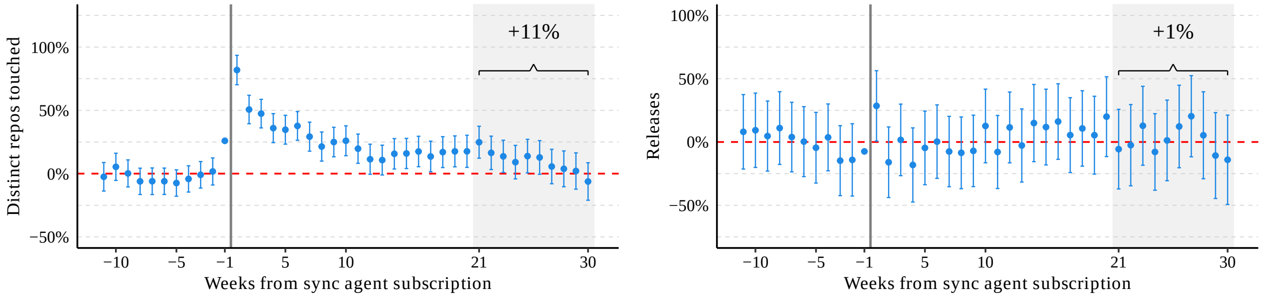
(a) Lines Changed

(b) Distinct Files



(c) Commits

(d) Pull Requests Created

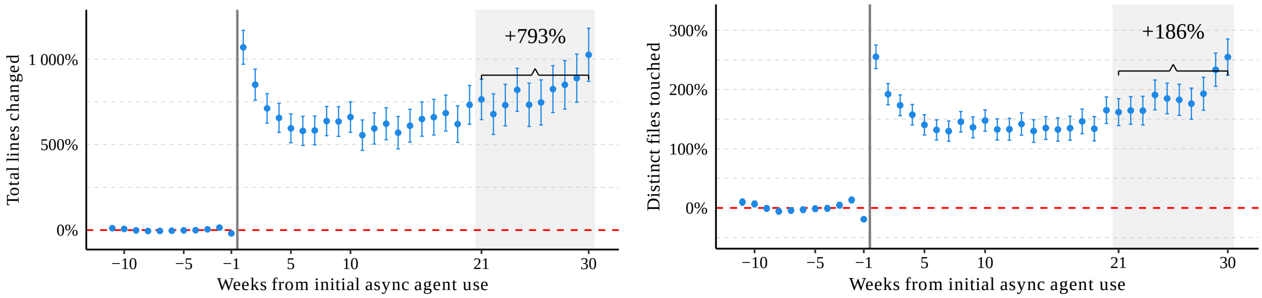


(e) Distinct Repositories

(f) Releases

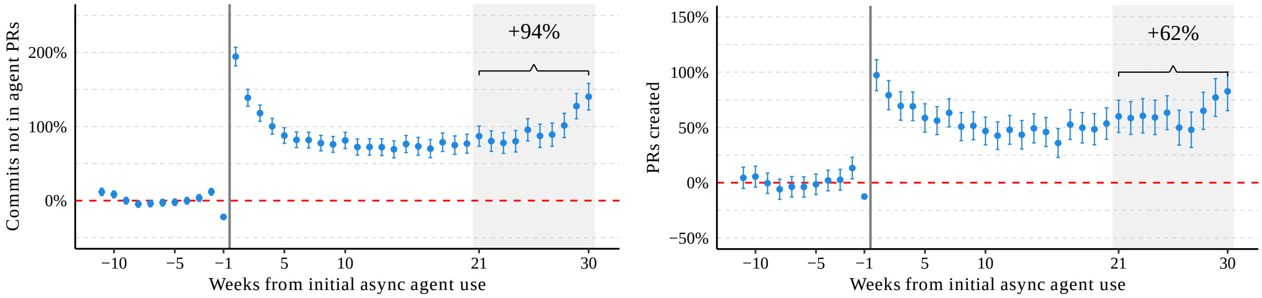
Notes: This figure presents matched event-study estimates of the effect of GitHub Sync Agent adoption across production layers, replicating Figure 8 for the individual tool. Outcomes are normalized by each developer's pre-period mean. The first post-adoption week is omitted as transitory effects distort the scale; the omitted coefficients are 1,565.9%, 341.4%, 200.9%, 5.4%, 2.5%, and −31.8% for Panels OA-6(a)–OA-6(f), respectively.

Figure OA-7: OpenAI Codex Sync Effect Across Production Layers



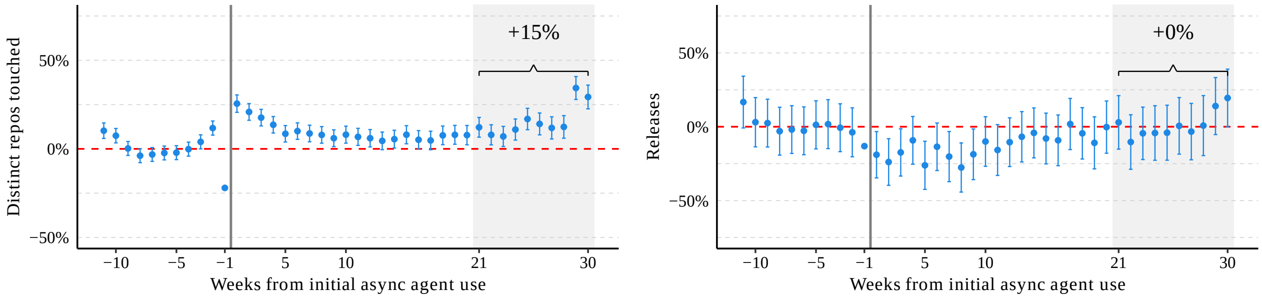
(a) Lines Changed

(b) Distinct Files



(c) Commits (Human-Authored)

(d) Pull Requests Created

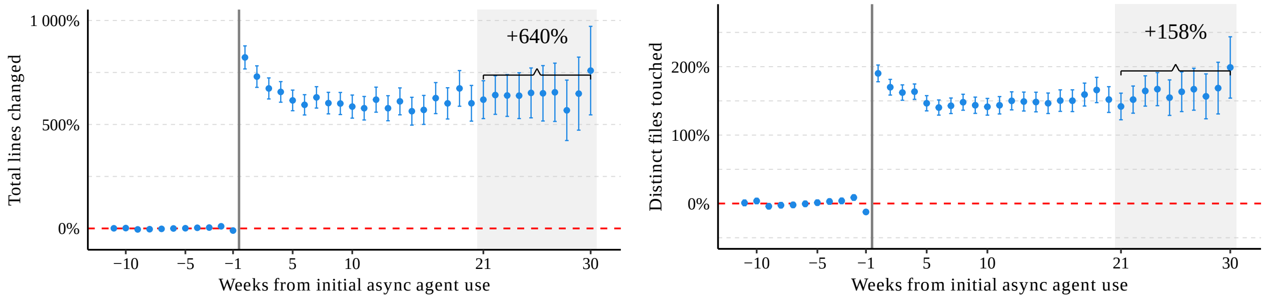


(e) Distinct Repositories

(f) Releases

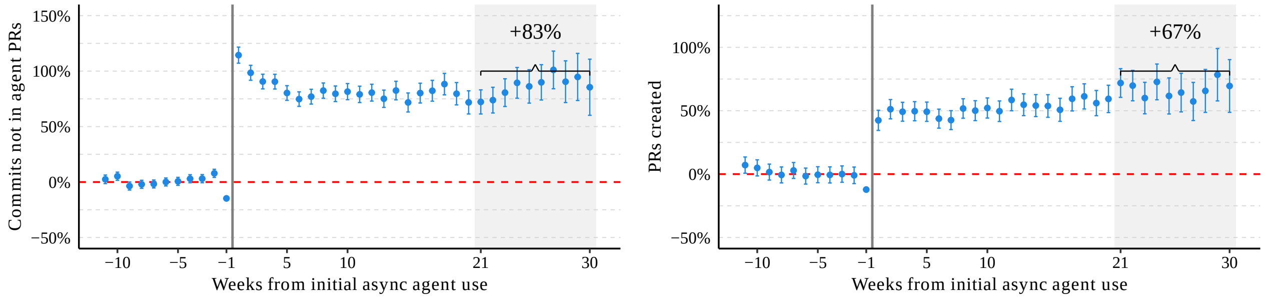
Notes: This figure presents matched event-study estimates of the sync-component effect of OpenAI Codex adoption across production layers, replicating Figure 8 for the individual tool. Outcomes are normalized by each developer's pre-period mean. The first post-adoption week is omitted as transitory effects distort the scale; the omitted coefficients are 1,948.1%, 432.9%, 288.3%, −78.9%, 26.2%, and −49.5% for Panels OA-7(a)–OA-7(f), respectively.

Figure OA-8: GitHub Async Agent Sync Effect Across Production Layers



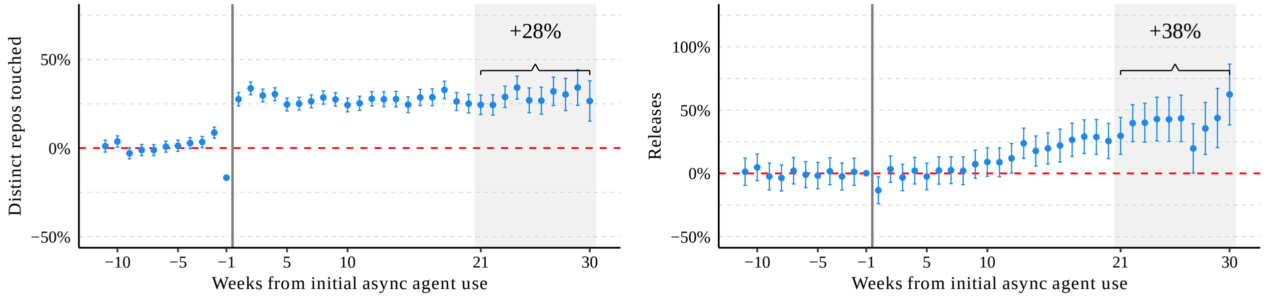
(a) Lines Changed

(b) Distinct Files



(c) Commits (Human-Authored)

(d) Pull Requests Created

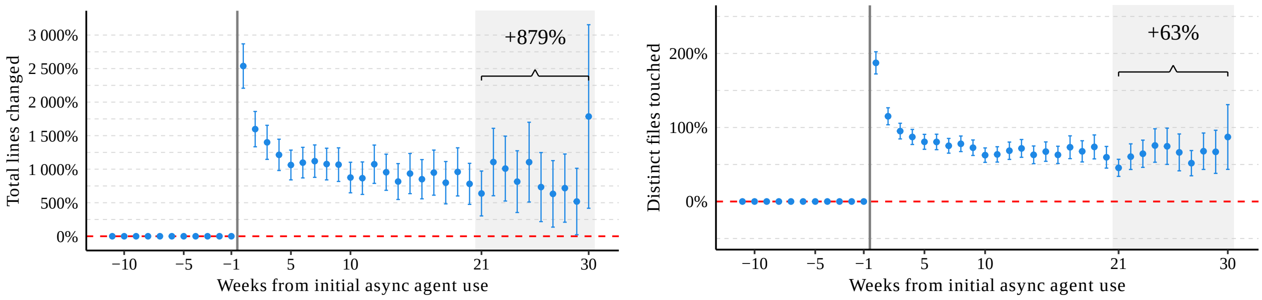


(e) Distinct Repositories

(f) Releases

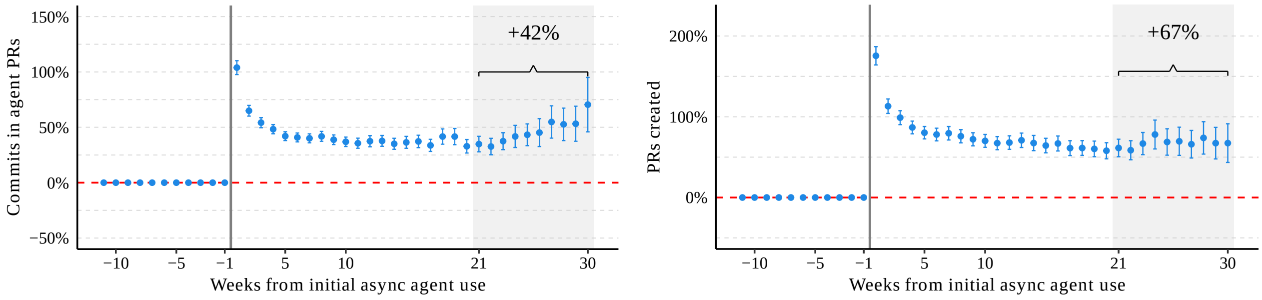
Notes: This figure presents matched event-study estimates of the sync-component effect of GitHub Async Agent adoption across production layers, replicating Figure 8 for the individual tool. Outcomes are normalized by each developer's pre-period mean. The first post-adoption week is omitted as transitory effects distort the scale; the omitted coefficients are 1,293.0%, 254.4%, 141.1%, −189.0%, 0.9%, and −21.8% for Panels OA-8(a)–OA-8(f), respectively.

Figure OA-9: GitHub Async Agent Async Effect Across Production Layers



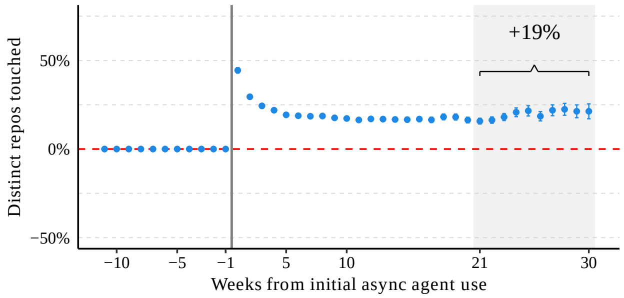
(a) Lines Changed

(b) Distinct Files



(c) Commits (Agent-Authored)

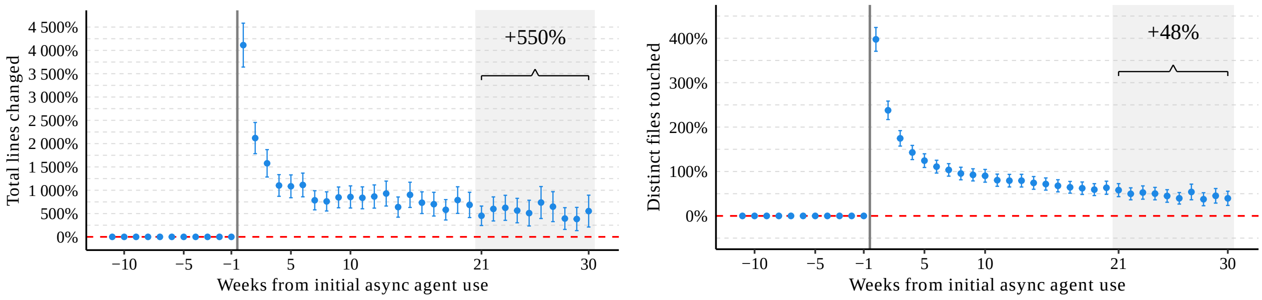
(d) Pull Requests Created



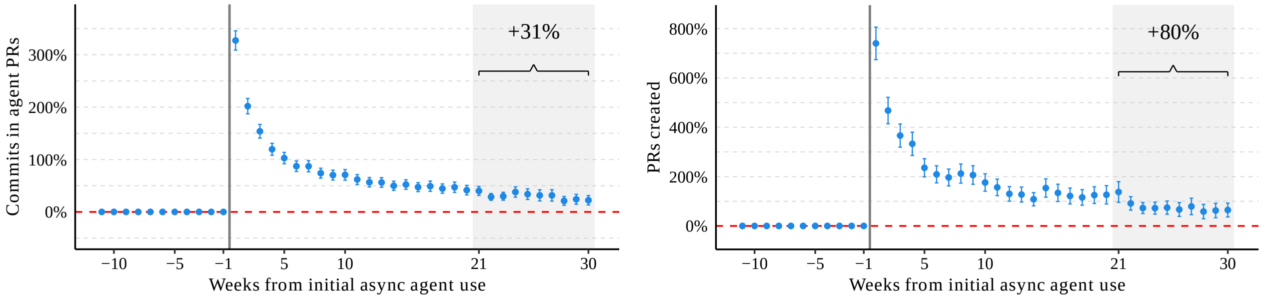
(e) Distinct Repositories

Notes: This figure presents interrupted time-series estimates of the async-component effect of GitHub Async Agent adoption across production layers, replicating Figure 8 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 12,510.9%, 777.6%, 372.4%, 653.5%, and 224.1% 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.

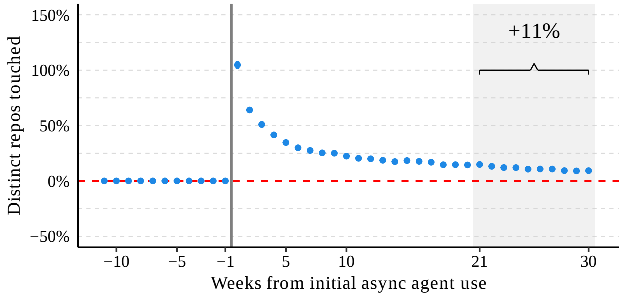
Figure OA-10: OpenAI Codex Async Effect Across Production Layers



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



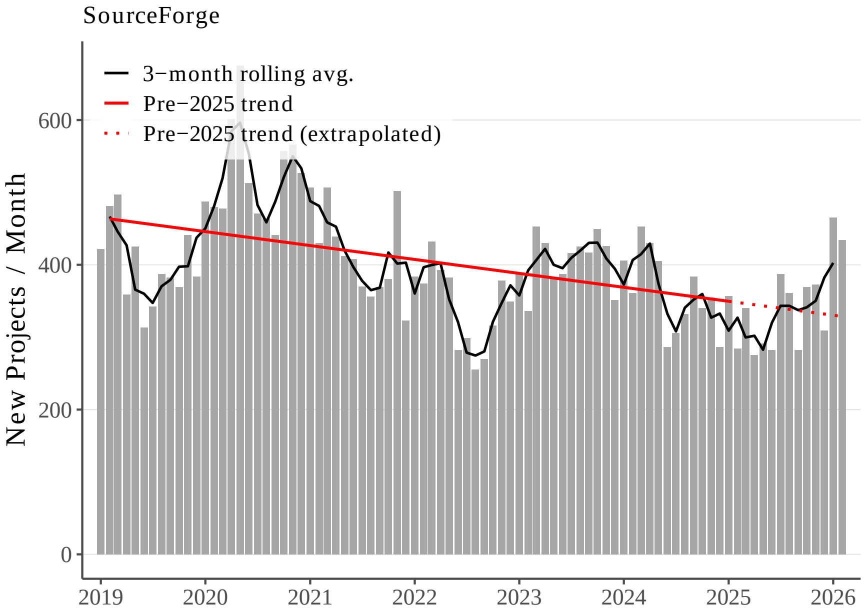
(c) Commits (Agent-Authored) **(d)** Pull Requests Created



(e) Distinct Repositories

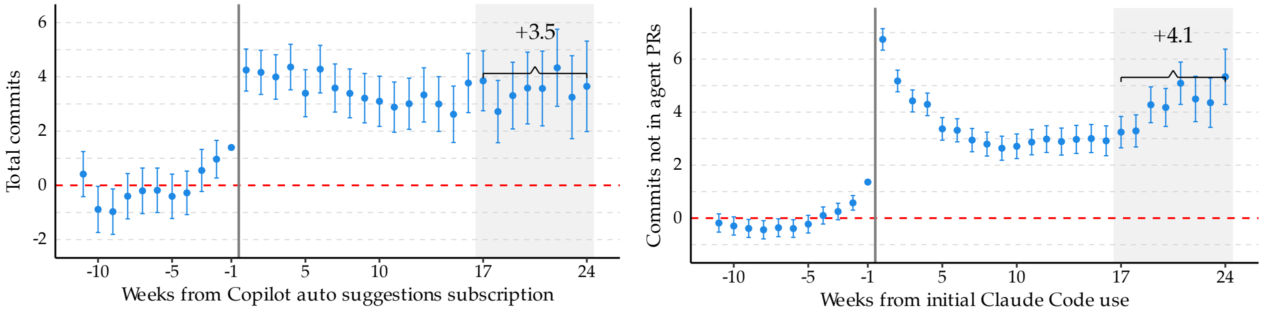
Notes: This figure presents interrupted time-series estimates of the async-component effect of OpenAI Codex adoption across production layers, replicating Figure 8 for agent-authored output. Outcomes are normalized by each developer's pre-period mean. The first post-adoption week is omitted as transitory effects distort the scale; the omitted coefficients are 15,656.3%, 1,252.6%, 817.2%, 1,710.0%, and 437.0% 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: New Projects on SourceForge

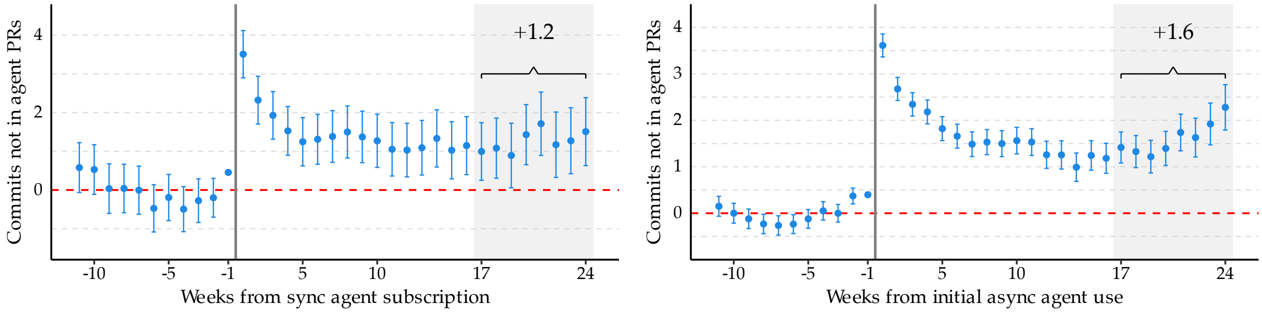


Notes: Monthly count of newly created SourceForge projects from 2019 through early 2026. Bars are raw monthly values; the solid black line is a 3-month rolling average. Unlike the iOS App Store and the Chrome Web Store (Figure 12), SourceForge shows no acceleration in new releases around the rise of agentic coding tools.

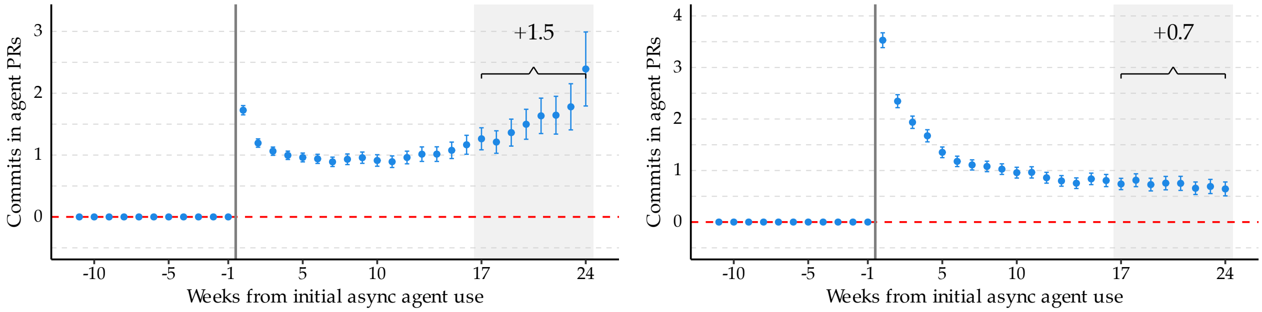
Figure OA-12: Productivity Effects by Tool (Levels)



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



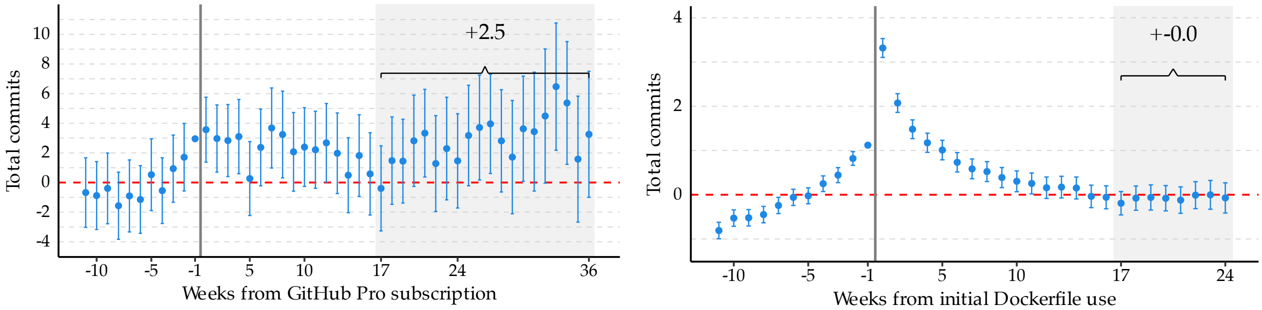
(c) GitHub Sync **(d)** OpenAI Codex (Sync)



(e) GitHub Agent (Async) **(f)** OpenAI Codex (Async)

Notes: This figure presents the same per-tool event studies as Figure OA-3, but with the outcome measured in levels (raw commit counts) rather than normalized by the pre-period mean.

Figure OA-13: Productivity Effects of Placebo Tools (Levels)



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

Notes: This figure presents level-effect event studies for two non-AI coding tools included as placebo comparisons. GitHub Pro is a paid subscription providing collaboration and compute features; Docker adoption is identified through the appearance of a Dockerfile in a repository. No detectable level effect is found for either tool.

Table OA-5: Treatment 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 D2–D9	Weight. D2–D9	Pooled Norm.
Lines Changed	13,108.7	1985.0	835.0	341.9	220.3	131.1	91.3	51.9	14.2	−3.1	458.8	400.2	741.3
	(1731.3)	(221.4)	(82.1)	(37.6)	(21.6)	(12.7)	(8.3)	(4.6)	(2.5)	(0.3)	(30.4)	(25.1)	(19.2)
Distinct Files	2075.9	552.9	203.9	136.9	96.0	59.1	44.5	29.9	20.9	10.6	143.0	125.2	187.0
	(186.1)	(43.2)	(17.4)	(10.6)	(6.8)	(4.6)	(3.5)	(2.5)	(1.8)	(1.0)	(6.2)	(5.1)	(4.3)
Commits	1284.6	380.9	195.5	121.6	91.1	66.4	37.0	36.0	25.2	11.6	119.2	101.8	109.1
	(122.2)	(32.8)	(16.7)	(9.9)	(7.1)	(5.2)	(3.3)	(2.5)	(1.8)	(0.8)	(5.0)	(3.9)	(2.7)
PRs Created	262.2	145.9	108.5	88.4	85.8	62.5	49.4	36.4	25.9	8.7	75.3	72.3	65.5
	(28.4)	(16.7)	(12.3)	(9.3)	(7.3)	(5.8)	(4.5)	(3.7)	(2.7)	(1.4)	(3.3)	(3.0)	(2.3)
Distinct Repos	81.6	50.2	41.5	39.6	23.2	27.2	17.7	17.1	11.4	7.2	28.5	26.2	25.5
	(11.2)	(6.4)	(5.0)	(3.3)	(2.8)	(2.0)	(1.7)	(1.5)	(1.1)	(0.7)	(1.3)	(1.0)	(1.0)
Releases	59.0	53.6	28.6	46.3	35.4	18.9	17.2	16.7	9.7	−0.7	28.3	27.5	20.3
	(16.0)	(12.7)	(7.6)	(9.6)	(6.6)	(4.6)	(5.6)	(4.3)	(3.3)	(2.1)	(2.7)	(2.6)	(2.6)

Notes: Each cell shows the weeks 21–30 treatment effect from a level regression, normalized by the decile-specific mean of the outcome's treated-arm pre-period value and expressed in percent. Matched pairs are split into deciles using the pre-period mean of that row's mean outcome across treated and control, avoiding regression-to-the-mean. Differences between treated and control are winsorized at the 1st and 99th percentiles before estimation. *Raw D2–9*: unweighted average of decile effects D2 through D9, excluding the extreme tails. *Weighted D2–9*: each decile weighted by its share of users with a non-missing outcome at week 20, down-weighting high-attrition (typically low-activity) deciles like our main event-studies, again excluding D1 and D10. *Pooled Normalized*: normalized event-study coefficient from a regression on the per-user-normalized outcome (each user divided by own pre-mean before estimation).

References for Online Appendix

Jones, C. I. (2026). A.I. and Our Economic Future. *NBER Working Paper, No. 34779*.