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

Mert Demirer† Leon Musolff‡ Liyuan Yang§

March 26, 2026

Abstract

We study how productivity effects evolve across successive generations of AI coding tools and across stages of the software production process, using data on more than 100,000 GitHub developers combined with internal telemetry. In an event-study design with matched controls, we find that each successive generation of AI coding tools—autocomplete, interactive coding agents, and autonomous coding agents—significantly increases coding activity, with respective cumulative effects of 50%, 140%, and 180%. However, these effects attenuate sharply across production stages: for lines of code, the effect is 1,480%, but falls to 180% for commits, and 36% for releases. We show that this pattern is consistent with a multi-stage production process in which AI automates code generation but downstream tasks—review, integration, and coordination—remain human-bottlenecked. Aggregate data from three software marketplaces corroborate this result: large developer-level gains in task productivity only translate into modest increases in software output.

JEL: D22, D24, L86

Keywords: generative AI, productivity, software development, task-based automation, multi-stage production

*We thank Frank Nagle, Max Schnidman, Sida Peng and the METR team. This project was funded in part by the The Mack Institute For Innovation Management at Wharton.

†MIT, mdemirer@mit.edu

‡Wharton, lmusolff@wharton.upenn.edu

§MIT, lyyang@mit.edu

1 Introduction

A distinguishing feature of large language models (LLMs), relative to earlier technologies, is the speed of technological progress and diffusion. As a result, model capabilities have improved rapidly, and firms have deployed LLMs across a growing range of applications. While much of the existing literature emphasizes benchmark-based model performance or the productivity effects of individual tools, 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 LLM-based tools? Second, to what extent do productivity gains at the task level translate into final output—or are they attenuated by bottlenecks at other stages of production that remain unautomated?

We study these questions in the context of software development, one of the earliest and most prominent domains of LLM adoption and among the occupations most exposed to LLMs (Eloundou et al., 2024). Software development provides an ideal setting for two reasons. First, it has experienced multiple waves of widely adopted LLM-based tools. These include *autocomplete* functionality, *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 for hours 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.

We formalize this sequential structure in a multi-layer production model in which AI coding tools can affect each layer of the production process in one of two ways: by partially automating it (rendering human effort at that layer unnecessary but still requiring human review to integrate the output into downstream layers) or by augmenting it (making each unit of human effort more productive at that layer). 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 entire pull requests—which still require human review before they can be merged. A key prediction of the model is that when human effort at one layer is complementary to upstream input, even unbounded automation of the upstream layer yields only finite productivity gains—an implication reminiscent of the weak-links literature (Kremer, 1993; Jones, 2011). The model thus emphasizes human input as the mechanism that links successive stages of production, and predicts that productivity gains attenuate through the production hierarchy as output passes through stages that remain human-bottlenecked.

To test these predictions, we use comprehensive data on more than 100,000 developers from public GitHub repositories, together with internal Microsoft data covering multiple

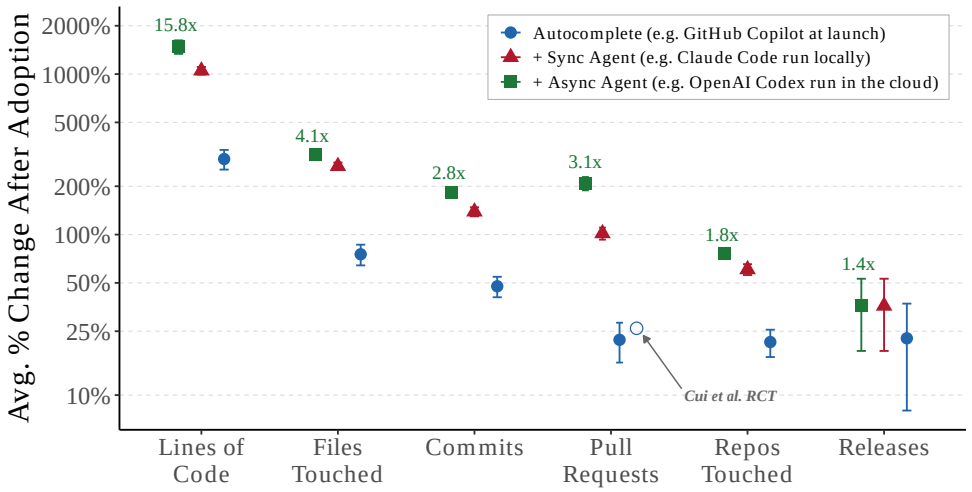


Figure 1: Overview of Productivity Effects of AI Coding Tools.

Note: 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 typically adopted alongside earlier generations, the figure presents cumulative effects of adopting all tools up to and including a given generation. "Cui et al. RCT" refers to Cui et al. (2025), who estimate the productivity effects of GitHub Copilot using a field experiment.

products and firms. We analyze how productivity effects evolve across generations of tools and across stages of production, and ask whether these developer-level gains translate into aggregate software output. We find large productivity effects that grow across successive generations of tools but attenuate sharply across stages of production—a pattern predicted by our model—and present suggestive evidence that these large gains only translate into modest increases in software output.

Our empirical analysis combines public GitHub data with internal Microsoft telemetry to identify when developers first adopt each tool—autocomplete (via Copilot subscriptions in 2022), sync agents (via Copilot agent mode and `CLAUDE.md` files), and async agents (via agent-authored pull requests from GitHub 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 who does not adopt the product but is active in the same calendar week one year earlier, avoiding contamination from widespread private adoption of AI tools among contemporaneous non-adopters. To mitigate activity bias, which may arise when adoption follows the start of a new project, we require control users to have at least one pull request on the treated user's adoption date and match users based on their pre-treatment activity over the 12 weeks preceding adoption.

Our results are summarized in Figure 1. We find that AI coding tools substantially

increase developer productivity, with effects that grow across successive generations of tools. Focusing on the number of commits, autocomplete increases developer productivity by approximately 50%; 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. Our autocomplete estimates are consistent with prior field-experimental evidence (Cui et al., 2025).

Our results also vary across measures of productivity in a way that matches the attenuation pattern predicted by our production model. Effects are largest for the most granular measures—such as lines of code—and decline monotonically as we move toward measures closer to final output: commits, pull requests, and ultimately releases, which the effect is much smaller. As the model suggests, AI has dramatically reduced the cost of the earliest stage of production—writing code—but the gains are progressively compressed at each downstream stage where human review remains required. The binding constraint on developer productivity appears to be shifting from writing code to reviewing and integrating it. Whether future generations of AI tools can ease this constraint—by producing higher-quality code that requires less review, or by automating the review process itself—remains an open question.

Finally, if these developer-level patterns hold in the aggregate, we would expect to see a clear acceleration in platform-wide coding activity but more muted effects on actual software output. This is broadly what we find. Aggregate coding activity on GitHub has accelerated sharply since early 2025, whether measured by the creation of new repositories, the number of active repositories, or the number of merged pull requests. These aggregate effects are plausible given the scale of adoption: our estimates suggest that Claude Code alone accounts for more than 5% of commits in early 2026.

Whether this increase in coding activity has translated into increased software output—either new applications on the extensive margin or more frequent and feature-rich updates on the intensive margin—is less clear. Using data from three software marketplaces (VS Code Extensions, SourceForge, and the Apple App Store), we find mixed results. Growth in new listings has accelerated for VS Code Extensions and, less dramatically, for iOS apps, but not for SourceForge. We find some suggestive evidence that update frequency and feature additions are increasing for VS Code Extensions, though such changes could also reflect AI-generated demand for new developer tooling rather than a pure supply-side effect.

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. Section 6 presents the productivity estimates. Section 7 examines aggregate software output. Section 8 concludes.

2 Literature

Our work relates to three strands of literature: AI and productivity, task-based automation and multi-stage production, and the debate over whether task-level productivity gains translate into aggregate output growth.

A growing literature studies the effects of generative AI tools on productivity. Initial studies using controlled lab-like experiments find significant productivity gains of 20–50% (Peng et al., 2023; Campero et al., 2022; Noy and Zhang, 2023; Dell'Acqua et al., 2023).[1] More recent work provides observational (Brynjolfsson et al., 2025) and field-experimental (Cui et al., 2025) estimates of similar magnitude. 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 observational evidence from corporate adopters suggests a 39% increase in productivity (Sarkar, 2025). We contribute 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. 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).[2] 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.

Finally, our aggregate analysis connects to a longstanding debate about whether productivity gains at the micro level translate into aggregate output growth. Solow (1987)

[1] Vaithilingam et al. (2022) is a notable exception.

[2] Relatedly, Agrawal et al. (2018) frame AI as a drop in the cost of prediction, which raises the value of complementary human judgment. In our setting, code generation is the prediction task that AI has cheapened, while review and integration are the judgment tasks that remain human-supplied.

famously noted the disconnect between rapid IT investment and sluggish productivity statistics, and Acemoglu (2025) has argued that AI's aggregate impact may be modest if it automates tasks that account for a small share of economic value. Brynjolfsson et al. (2021) offer a possible explanation: general-purpose technologies require complementary organizational investments before their productivity gains manifest in aggregate 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.

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. Developers may work in parallel across multiple branches and multiple projects (*repositories*). Finally, developers can create *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 large language models (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 four categories: autocomplete, chatbot interfaces, 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., 2025).

AI Chatbots Following the launch of ChatGPT in November 2022, developers 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.[4] Because chatbot usage spans many platforms and cannot be cleanly separated from non-coding activity without access to prompts, we do not analyze chatbots as a separate category in this paper.

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

[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] For GitHub Copilot, the public beta became available to business users in July, and chat functionality left the beta phase in December 2023.

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.

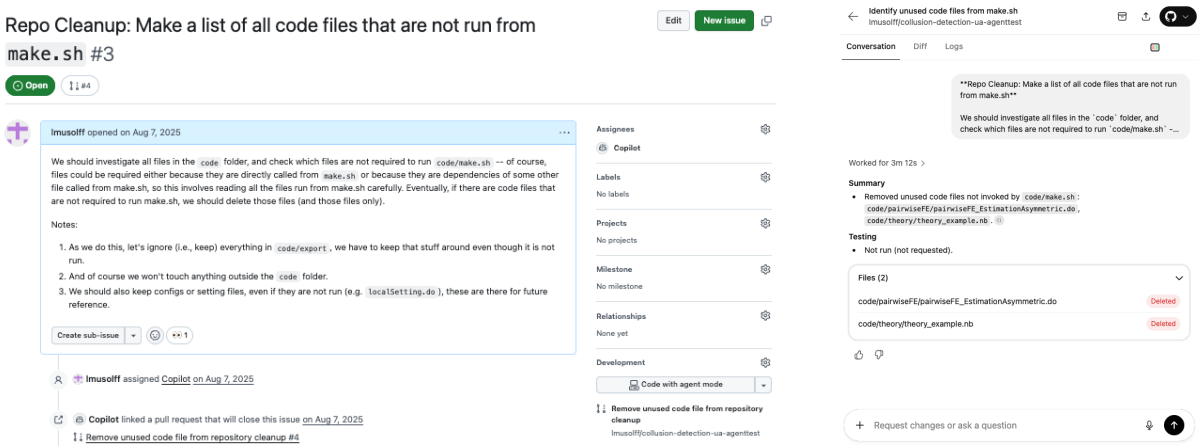
Async Agent A more recent development is the emergence of async coding agents. Unlike sync agents, which operate interactively in the IDE and require frequent developer input, async agents are designed to operate autonomously for extended periods with little or no ongoing human interaction. Developers 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-7 for an example.

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

[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.

[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.



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

Figure 2: Assigning Tasks to Async Coding Agents

Note: 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.

interface for Codex). Figure 2(b) shows an example of a developer assigning a task to Codex, and Appendix Figure OA-8 displays the resulting pull request.

Two features of the workflow and tools described above are central to our analysis. First, software production is inherently sequential: code must be written, organized into commits, reviewed in pull requests, and integrated into releases before it reaches users. Second, successive generations of AI tools operate at progressively higher layers of this hierarchy—from suggesting individual lines (autocomplete) to autonomously creating pull requests (async agents). We now develop a simple model that formalizes these features and derives predictions about how productivity gains propagate—or attenuate—across production stages.

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 shipped apps—and how successive generations of AI tools intervene at different points in this pipeline. At each layer, many lower-level units must be combined, reviewed, and refined—a process that requires human effort. 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

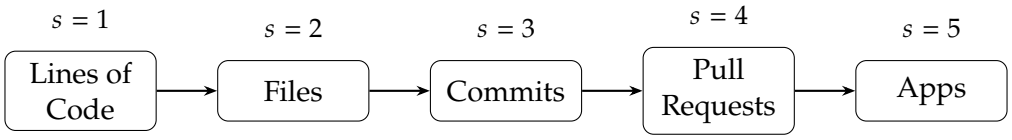


Figure 3: The Software Production Hierarchy

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, output is:

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

where e_1 is the effective developer input devoted to writing code. At each subsequent layer $s \geq 2$, production combines upstream input with effective effort 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 developer input at layer s (defined below), $\alpha_s \in (0, 1)$ is the share parameter, and $\sigma_s > 0$ is the elasticity of substitution between upstream input and developer input. Final output is $Y \equiv y_S$.

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

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

where $h_s \geq 0$ 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), $B_s \geq 1$ is a productivity parameter that captures how much AI increases the output the developer produces per unit of effort, $a_s \geq 0$ is the quantity of AI-generated output at layer s, and $\phi_s \geq 0$ governs how much AI output a unit of human effort can absorb (review, validate, integrate). In the baseline without AI, $B_s = 1$ and $a_s = 0$ for all s, so $e_s = h_s^p$: all input is from the developer. We refer to e_s as "developer input" throughout, whether it originates from the human, the AI, or both.

The two terms capture distinct channels through which AI affects production:

- **Augmentation** ($B_s h_s^p$): AI makes the human more productive at layer s—for example, by suggesting code completions or highlighting potential issues during review. The human remains essential; if $h_s^p = 0$, the augmentation channel contributes nothing.
- **Partial automation** ($\min(a_s,\ \phi_s h_s^r)$): 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 implies that for each unit of AI output, the developer must spend time proportional to $1/\phi_s$ reviewing it; unreviewed AI output is not productive. The parameter ϕ_s reflects AI quality at layer s: higher-quality AI output requires less review effort per unit, so ϕ_s is larger.
- **Full automation** ($\phi_s \to \infty$): AI output requires no human review and $\min(a_s, \phi_s h_s^r) \to a_s$. The human is no longer needed at layer s.

AI's productivity impact. Although augmentation, human oversight, and full automation are distinct mechanisms, their effect on the model is similar: each increases the productivity of generating output at a given layer, as the same output can be produced with less human effort. They differ in magnitude. Under full automation, the productivity gain is in principle infinite: the effective input is a_s, which can be scaled up at the cost of inference alone. Under augmentation, the gain is proportional to B_s, which depends on how much AI improves the developer's own productivity. Under human oversight, the gain depends on both the quantity of AI output a_s and the AI quality parameter ϕ_s, which determines how much review effort each unit of AI output requires.

Interpretation. The model captures several 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—augmenting human productivity (B_s) or generating output that requires human oversight (a_s)—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 from the production process: even when AI generates output autonomously, the Leontief structure ensures that unreviewed AI output contributes nothing. The developer's scarce resource is not code-writing ability but *attention*—the capacity to review, validate, and integrate AI-generated work.

Generality. While we focus on the specific layers of software production (code, files, commits, pull requests, apps), the framework is more general. Some layers may correspond to creative and design work—for instance, architectural decisions about how to structure a codebase, or the initial idea for a feature. These layers are analogous to the "idea production

function" in the growth literature (Aghion et al., 2019; Jones and Tonetti, 2026), and the same logic applies: AI can generate candidate designs or feature proposals ($a_s > 0$), but human judgment is required to evaluate and select among them (ϕ_s finite).

4.2 Mapping AI Coding Tools to the Production Function

The framework above is general. 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 can replace ($a_s > 0$), and which layers remain purely human.

Autocomplete. Autocomplete tools (e.g., GitHub Copilot's inline suggestions) propose code snippets as the developer types, helping her write lines of code faster. The developer retains full control: she accepts, modifies, or rejects each suggestion. In the model, autocomplete operates exclusively through the augmentation channel at layer 1:

$$e_1 = B_1 h_1, \qquad B_1 > 1, \qquad a_1 = 0. \tag{4}$$

The developer still writes every line, but each unit of human effort h_1 produces more output. Since autocomplete does not operate at any higher layer, all developer input above layer 1 remains purely human ($B_s = 1$, $a_s = 0$ for $s \geq 2$). The gain in code generation must therefore pass through every subsequent layer of human-bottlenecked aggregation before reaching final output.

Sync agents. Sync agents (e.g., Claude Code, Cursor Agent) can autonomously generate and edit code across multiple files, but the developer monitors the process in real time. The agent generates code and assembles files directly ($a_1, a_2 > 0$), but the developer must review and approve the output—the Leontief term binds:

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

The developer's role at these layers shifts from writing code to reviewing AI-generated code. The AI produces the raw output (a_s), but each unit of that output requires human review time proportional to $1/\phi_s$ before it becomes productive. At the higher layers, the agent does not generate commits or pull requests directly, but it can therefore increase the developer's productivity in these layers—for example, by summarizing changes, documenting code, or highlighting relevant context for review:

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

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)	$s = 5$ (Apps)
Autocomplete	Augmenting	—	—	—	—
Sync agent	Partial automation	Partial automation	Augmenting	Augmenting	—
Async agent	Partial automation	Partial automation	Partial automation	Partial 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.

Relative to autocomplete, the sync agent operates through both channels simultaneously: it generates output at layers 1–2 ($a_s > 0$, bottlenecked by the developer's review capacity) and raises the developer's productivity at layers 3–4 ($B_s > 1$).

Async agents. Async agents (e.g., GitHub Copilot Coding Agent, Claude Code in headless mode) operate autonomously for extended periods. 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 generates 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 $a_s > 0$ at more layers—the async agent produces commits (a_3) and pull requests (a_4) autonomously, not just code and files. However, the Leontief structure still applies: at every layer, the developer must review the agent's output before it becomes productive. At the app layer, the developer reviews and decides whether to integrate the agent's pull requests into the shipped product. Human judgment at this final layer remains the binding constraint. This is consistent with the empirical finding that agent-authored pull requests have non-trivial rejection rates: the human review bottleneck ($\phi_4 h_4$) compresses the agent's contribution to final output.

Summary. Table 1 summarizes how each generation of AI coding tools maps to the model's parameters.

Three patterns emerge from Table 1. First, each successive generation generates AI output ($a_s > 0$) at progressively more layers: autocomplete does not autonomously generate AI output, but sync agents generate output at layers 1–2, and async agents at layers

1–4. Second, most generations also *augments* human productivity at the layers where the developer remains essential—autocomplete at layer 1, agents at layers 3–4. Third, for every tool and every layer where AI generates output, the developer's review capacity ($\phi_s h_s$) constrains how much of that output is usable. The human review bottleneck—not AI generation capacity—limits how much AI-generated output translates into shipped software.

4.3 The Elasticity of Substitution and Shock Propagation

The parameter σ_s in the production function (2) governs how easily upstream output can substitute for developer input at layer s. Three cases are of particular interest:

- **Complements** ($\sigma_s < 1$): upstream input and developer input must be used in rough proportion. An abundance of code is of limited value without commensurate review effort. In the limit $\sigma_s \to 0$, the technology approaches Leontief: $y_s \to \min(y_{s-1}, e_s)$.
- **Substitutes** ($\sigma_s > 1$): upstream input can partially replace developer input. Abundant code compensates for limited review capacity.

This distinction determines how gains at one layer propagate to the layers above. Suppose AI increases developer input at some layer k, raising output at that layer by a factor $A > 1$. Holding developer input fixed at every other layer, how much of this gain reaches final output? The pass-through at each layer s above the shock is governed by the output elasticity with respect to upstream input:

$$\theta_s \equiv \frac{\partial \ln y_s}{\partial \ln y_{s-1}} = \frac{\alpha_s \, y_{s-1}^{\frac{\sigma_s - 1}{\sigma_s}}}{\alpha_s \, y_{s-1}^{\frac{\sigma_s - 1}{\sigma_s}} + (1 - \alpha_s) \, e_s^{\frac{\sigma_s - 1}{\sigma_s}}}. \tag{8}$$

At a single layer, the exact pass-through of a shock of size A is (see Appendix A for the derivation):

$$\frac{\tilde{y}_s}{y_s} = \left[\theta_s \, A^{\rho_s} + (1 - \theta_s)\right]^{1/\rho_s}, \qquad \rho_s = \frac{\sigma_s - 1}{\sigma_s}. \tag{9}$$

This formula depends only on θ_s and σ_s, not on the structural parameter α_s separately. Three results follow.

Proposition 1 (Attenuation)**.** *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. At each intervening layer, only a fraction of the gain passes through. 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 developer input whose quantity has not changed. Since $\theta_s < 1$, 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.

Proposition 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: async agents (entering at layer 4) face only one layer of attenuation before reaching apps, while autocomplete (entering at layer 1) faces four. Even if autocomplete produces a larger raw increase in code, the async agent is predicted to have a larger effect on shipped software.

Proposition 3 (Complements versus substitutes)**.** *The elasticity of substitution σ determines both the local and global properties of pass-through:*

1. ***Shape of attenuation across layers.*** *Consider a shock propagating through multiple layers with common σ and θ, and let G_s denote the cumulative output gain after s layers of pass-through. The sequence $\{\ln \ln G_s\}$ is strictly convex in s when $\sigma < 1$ (complements), linear when $\sigma = 1$ (Cobb–Douglas), and strictly concave when $\sigma > 1$ (substitutes). With complements, attenuation is front-loaded: the first bottleneck layer absorbs a disproportionate share of the gain, and subsequent layers matter progressively less. With substitutes, attenuation accelerates through the hierarchy.*

2. ***Full automation.*** *When upstream input grows without bound ($A \to \infty$), the output gain converges to:*

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

   With complements, even infinite upstream input yields bounded gains—no amount of AI-generated code can overcome the scarcity of human review effort. With substitutes, full automation translates into unbounded output gains.

Proposition 3 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 ($\sigma < 1$), automating some

tasks has bounded effects on aggregate output—total production is constrained by the weakest link. 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 $\sigma = 1/2$.

Our model applies this logic *vertically* within a single production process. The "tasks" are the layers of the production hierarchy, and the weak link is the human-bottlenecked layer with the least capacity. The complementarity between layers means that even infinite code generation cannot overcome limited human review capacity. In the Leontief limit ($\sigma_s \to 0$), each layer becomes $y_s = \min(y_{s-1}, e_s)$, and output is determined entirely by the layer with the least developer input—the bottleneck is absolute.

5 Data

5.1 Data Sources

Our analysis draws on two data sources: publicly available GitHub data on activity in public repositories and Microsoft-internal data on the adoption and usage of AI-based coding tools available in GitHub Copilot. Detailed data collection procedures are in Appendix B.

Github Data GitHub is the dominant platform for hosting and collaborating on software projects, with over 100 million developers and 400 million repositories. Repositories can be either public or private; our data includes only activity on public repositories. 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.

We use GHArchive to construct a universe of active developers. GHArchive continuously records all events in public GitHub repositories and makes them publicly available. This data enables us to identify all active developers on the platform at scale without exhausting API rate limits. We use GHArchive primarily to construct a pool of potential control developers in our productivity event studies: we identify users whose first recorded activity occurred before January 2020 and whose most recent activity occurred after December 2022, ensuring a sample of developers with sustained engagement on the platform. This restriction yields approximately 5.5 million users.

For control developers selected using these criteria, as well as for developers who adopt AI coding tools (with selection procedures described below), we use the GitHub

REST API to download their complete commit and pull request histories. 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.

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. Our usage sample spans April to 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 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 these two products.

5.2 Construction of Adoption Events

Measuring adoption is challenging because detailed usage data are typically proprietary. We overcome this by combining 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 include users who adopted before the end of 2022 and end the sample in 2022 to avoid potential spillover effects from AI chatbots that became prevalent in 2023.

Sync Agents Using internal Microsoft data, we identify the first date each user invokes GitHub Copilot in agent mode. We define this date as the GitHub Sync Agent adoption event. To isolate sync agent usage from async agent usage, we restrict this sample to users who never use the GitHub Async Agent. Furthermore, 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.

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 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.[7] 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. We document descriptive patterns of adoption and usage of these tools in Appendix C.

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. 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 their release dates. For each tool, we construct a sample that begins in the calendar year of its initial release and runs through the end of that same calendar year. GitHub Autocomplete was released in May 2022; GitHub Sync Agent was released in February 2025; Claude Code was released in February 2025; GitHub Async Agent was released in May 2025; and Codex was released in May 2025. Accordingly, each tool's sample begins at the start of its release year and ends on December 31 of that year.

Because we obtain the data through API calls, our sample does not include coding activities that were later deleted by users. In addition, we collect information only from

[7] Adoption of an async agent does not imply exclusive use of that agent: as shown in Table **??**, 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.

Table 2: Pre-Adoption Summary Statistics for Adopters

Statistic	GitHub Autocomplete	Pooled Sync	Pooled Async
Lines Changed	42,467.77	25,905.54	26,994.77
(sd)	(461,797.17)	(658,287.76)	(717,240.67)
[p5, p95]	[0.00, 114,183.20]	[0.00, 68,838.17]	[0.00, 72,505.25]
Distinct Files	90.73	26.90	25.67
	(362.70)	(188.83)	(193.66)
	[0.00, 306.87]	[0.00, 108.55]	[0.00, 104.91]
Commits	17.82	5.79	5.46
	(50.35)	(21.09)	(20.18)
	[0.00, 65.31]	[0.00, 22.91]	[0.00, 21.55]
PRs Created	1.59	0.64	0.59
	(2.95)	(4.06)	(4.17)
	[0.00, 6.00]	[0.00, 3.09]	[0.00, 2.82]
Distinct Repos	3.62	0.79	0.74
	(10.14)	(2.37)	(2.06)
	[0.00, 13.27]	[0.00, 2.73]	[0.00, 2.55]
Releases	1.17	0.39	0.37
	(7.29)	(4.48)	(4.69)
	[0.00, 4.00]	[0.00, 1.27]	[0.00, 1.09]
N	14,873	159,927	133,084

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 row reports a weekly outcome averaged over the pre-adoption period (weeks −11 to −1 relative to adoption). The first line is the mean across adopters, followed by the standard deviation in parentheses and the 5th and 95th percentiles in brackets.

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.

We discuss our outcome variables as well as the sample construction in more detail in Appendix B.

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, 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, 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 Effects

We now examine the productivity effects of AI coding tools along two dimensions: across successive generations of tools, and across stages of the software production hierarchy.

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. 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.[8]

Another issue is that 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.

[8]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.

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 sample 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.

Our primary outcome variable is the percentage change in weekly commits, constructed by dividing a developer's weekly commit count by their pre-period average and winsorizing at the 99th percentile.[9] We drop developers with no commits in the pre-period. The effects we report should therefore 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.[10] This interpretation 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 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.

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.

[9] That is, we cap values above the 99th percentile at the 99th percentile value.

[10] For example, suppose treatment increases the output of less productive developers more. A developer who averages 1 commits 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%.

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. Therefore, to isolate the effect of async agents, we pool the sample of users of both async agents and use agent-authored commits as the outcome variable. This directly measures the productivity effect of async agents, as we observe their output. To estimate the impacts of sync agents, we pool samples from the users of both sync and async agents, but use human-authored commits from async-agent users. Again, the idea here is that async agents can increase observable output only through their own commits, so the change in human commits should come from sync agents.

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	13,991	5,173	16,904	55,474	39,851	1,184	39,472
Treated Group	3,017	1,108	1,736	1,698	1,243	2,320	934
Matched Control Group	1,229	502	524	506	309	1,054	258
Median Weekly Distinct Files							
# Treated Units	14,006	5,187	16,933	55,648	40,057	1,185	39,619
Treated Group	24.0	11.5	12.6	12.5	8.27	18.6	7.82
Matched Control Group	17.1	7.64	7.91	7.00	4.91	14.5	4.27
Median # Weekly Commits							
# Treated Units	14,096	5,253	17,388	57,567	41,303	1,190	40,435
Treated Group	6.18	2.82	3.18	2.73	1.91	5.32	1.73
Matched Control Group	6.18	2.82	2.64	2.27	1.55	5.32	1.45
Donor Pool	3.67	3.07	3.00	3.00	3.00	3.67	3.00
Median Weekly PRs Created							
# Treated Units	10,296	3,460	8,271	25,244	12,061	650	10,846
Treated Group	1.18	0.91	0.64	0.64	0.45	0.95	0.36
Matched Control Group	0.91	0.64	0.91	0.73	0.73	1.00	0.64
Median Weekly Distinct Repos							
# Treated Units	14,011	5,190	16,942	55,674	40,089	1,187	39,639
Treated Group	1.27	0.73	0.64	0.64	0.45	1.00	0.45
Matched Control Group	1.55	0.82	0.73	0.64	0.55	1.36	0.55
Median Weekly Releases							
# Treated Units	2,918	855	2,815	6,059	2,629	234	3,067
Treated Group	0.73	0.64	0.55	0.55	0.36	0.64	0.45
Matched Control Group	0.45	0.55	0.55	0.55	0.55	0.64	0.55
Sample Period Start Date	2022-06-21	2024-05-13	2025-02-24	2025-05-19	2025-05-16	2022-06-21	2025-01-01
Sample Period End Date	2022-12-26	2024-12-29	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).

6.2 Productivity Across Generations of Tools

We present our event studies in Figure 4 and the corresponding headline results in Table 4. The effect on commits is shown separately for each generation: autocomplete (all commits), sync agents (human-authored commits), and async agents (agent-authored commits).

The top row shows users who adopted GitHub Copilot in 2022, when only AI-based code completion was available. Our estimate in Figure 4(a) of a 48% increase in the number of commits is consistent with prior field-experimental estimates (Cui et al., 2025). The pre-trend is almost perfectly flat, supporting the parallel trends assumption. As shown in Figure 4(b), the effect is larger for less active developers, though it remains substantial throughout the activity distribution.

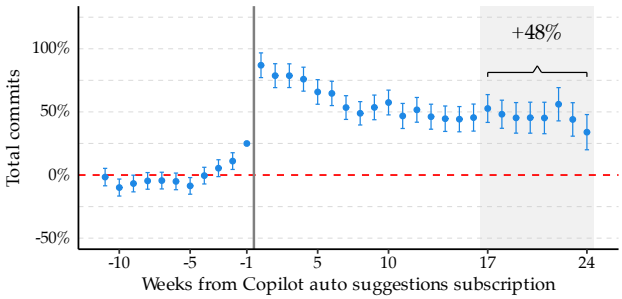
Moving on to sync agents in the middle row, we again find remarkably flat pre-trends and a sizable effect immediately upon adoption (Figure 4(c)). While the number of commits temporarily increases sharply—likely due to an experimentation phase—the effect rapidly decays before stabilizing at a 90% productivity gain—about double the effect of autocomplete. The heterogeneity pattern (Figure 4(d)) mirrors that of autocomplete: effects are largest for less active developers.

Finally, the bottom row shows the effect of adopting async agents on agent-written commits (Figure 4(e)). Agent-authored commits are zero by definition before adoption, and they increase by about 40% 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 4(f)).[11]

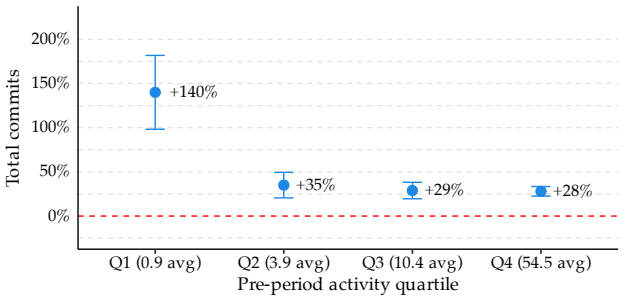
In Table 4, we break out the sync and async effects by product; the corresponding per-tool event studies are in Appendix Figure OA-10. Claude Code shows a notably larger sync effect than the other tools, which may reflect selection: Claude Code is primarily operated via a command-line interface, and its adopters may be more experienced coders who would benefit more from any tool. Among async agents, OpenAI Codex has a slightly larger effect than GitHub's agent.

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 50%; the cumulative effect including sync agents rises to approximately 140%; and additional use of async agents raises it to 180% through

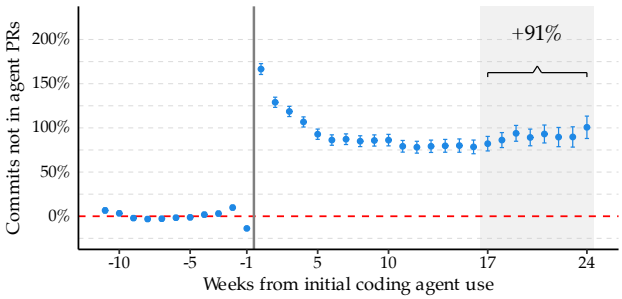
[11] In Appendix D, we examine learning dynamics for async agent users directly, finding that task descriptions assigned to the agent become longer over the first month and that merge rates increase shortly after adoption.



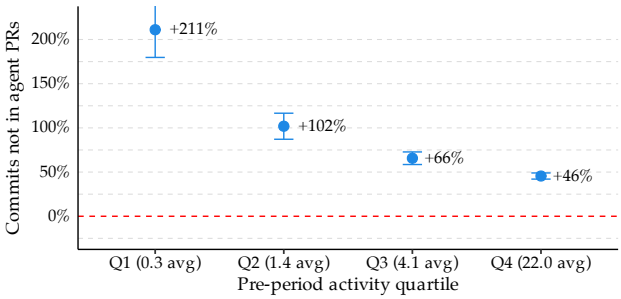
(a) Autocomplete: Event Study



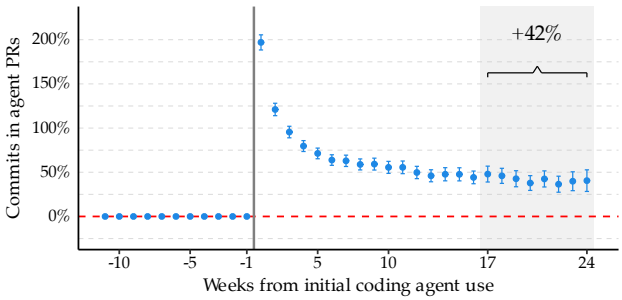
(b) Autocomplete: Heterogeneity



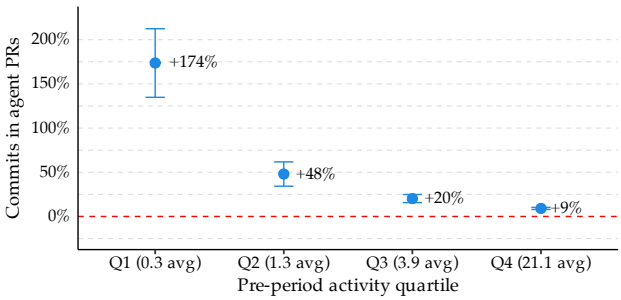
(c) Sync Effect: Event Study



(d) Sync Effect: Heterogeneity



(e) Async Effect: Event Study



(f) Async Effect: Heterogeneity

Figure 4: Productivity Effects of GenAI Coding Tools

Note: The left column presents matched event-study estimates of the effects of generative AI coding tools on commits; the right column shows treatment effects by pre-period activity quartile (weeks 17–24). Panel 4(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 4(c) pools adopters of sync agents (Claude Code, GitHub Sync, GitHub Agent, and OpenAI Codex) against matched controls. Panel 4(e) shows the effect on agent-authored commits for async agent adopters (GitHub Agent and Codex). Outcomes are measured relative to each developer's pre-period mean. The first post-adoption week is omitted as large transitory effects distort the scale; the omitted coefficients are 120%, 230%, and 555% for Panels 4(a)–4(e), respectively. In the right column, quartiles are assigned based on the average of both the treated and control developer's mean pre-period commits, to avoid regression to the mean.

Table 4: Change in Number of Commits by Coding Tool

	% Change in Number of Commits		
	Weeks 1–8	Weeks 9–16	Weeks 17–24
Autocomplete	70.5	49.5	47.6
	(2.7)	(2.9)	(3.5)
Sync Effect	114.9	83.1	91.3
	(1.8)	(2.0)	(2.9)
Claude Code	180.4	121.3	138.7
	(4.9)	(5.1)	(6.8)
GitHub Sync	117.9	67.5	70.8
	(6.5)	(6.0)	(7.7)
OpenAI Codex	110.3	73.0	86.3
	(3.5)	(3.9)	(4.9)
Async Effect	102.5	52.2	43.3
	(2.3)	(2.5)	(3.4)
GitHub Agent	56.5	32.3	31.3
	(1.6)	(1.8)	(2.8)
OpenAI Codex	156.3	68.4	49.8
	(5.0)	(4.4)	(5.1)

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 1-8' 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 17-24".

agent-authored commits. These effects are stable across tools within each category, and persist over at least 24 weeks. Having established that productivity effects increase across generations of tools, we now ask whether these gains propagate through the production hierarchy.

6.3 Productivity Across the Production Hierarchy

Our model predicts that productivity gains at the code-writing layer attenuate as they propagate through the production hierarchy, because each subsequent layer requires human effort that has not increased (Proposition 1). We now test this prediction by measuring the effect of AI tool adoption on outcomes at different layers of the hierarchy: lines of code, files, commits, pull requests, the number of distinct repositories a developer contributes to (a measure of whether developers take on more projects), and releases.

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. The effects are largest for lines of code, smaller for files and commits, smaller still for pull requests. The number of distinct repositories also increases, indicating that developers take on more projects, though the magnitude is much smaller than for lines of code or commits. For releases—the closest measure to final shipped output—the effects

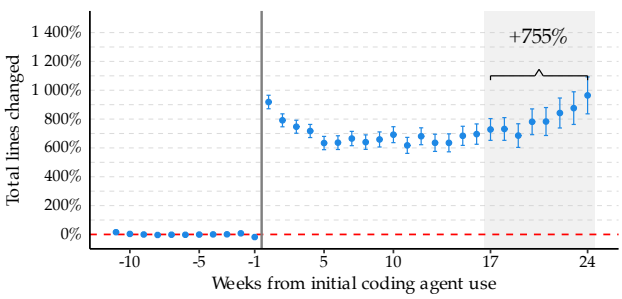
Table 5: Attenuation of Productivity Effects Across Production Layers

	Outcomes (Weeks 17–24, %)					
	Lines Chg	Dist Files	Commits	PRs	Dist Repos	Releases
Autocomplete	295.5	75.4	47.6	22.1	21.4	22.6
	(21.1)	(5.7)	(3.5)	(3.1)	(2.1)	(7.4)
Sync Effect	754.8	191.4	91.3	79.6	39.3	13.4
	(20.2)	(4.6)	(2.9)	(3.2)	(1.1)	(4.6)
Claude Code	747.3	219.6	138.7	94.5	51.0	38.3
	(41.4)	(9.9)	(6.8)	(6.8)	(2.5)	(10.5)
GitHub Sync	630.5	154.9	70.8	37.5	28.8	22.2
	(58.6)	(13.6)	(7.7)	(6.7)	(3.5)	(18.1)
OpenAI Codex	949.6	214.5	86.3	163.8	34.7	11.5
	(38.4)	(8.0)	(4.9)	(8.5)	(1.8)	(9.0)
Async Effect	429.1	47.2	43.3	106.8	15.0	—
	(66.6)	(3.9)	(3.4)	(9.1)	(0.6)	
GitHub Agent	364.4	42.0	31.3	83.1	15.7	—
	(99.6)	(5.0)	(2.8)	(7.1)	(0.8)	
OpenAI Codex	450.0	49.8	49.8	136.1	14.6	—
	(82.1)	(5.1)	(5.1)	(18.6)	(0.7)	

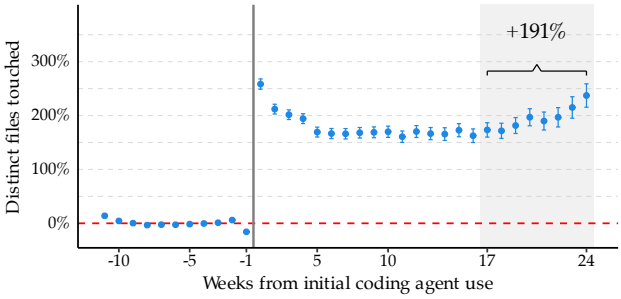
Notes: This table reports event-study coefficients (1% winsorized, weeks 17–24 bin, normalized by pre-period mean) measuring the effect of adoption of various Generative AI coding tools on outcomes spanning the production hierarchy. Lines Chg = weekly total lines changed, Dist Files = distinct files, 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.

are positive but substantially smaller than for other outcomes, and our power to detect effects is limited given the small sample of developers with pre-period release activity, as discussed above. This gradient is precisely the pattern predicted by Proposition 1: at each layer, the developer must review, coordinate, and integrate the output of the layer below—effort that AI has not reduced—so each human-bottlenecked layer absorbs a fraction of the upstream gain.

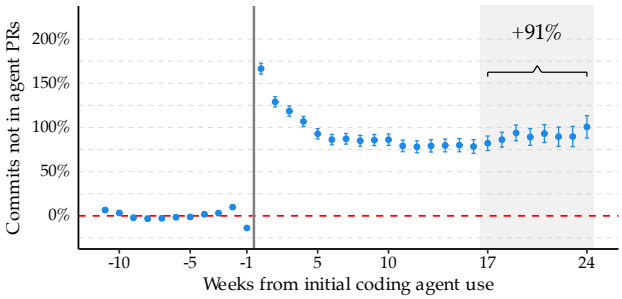
Figure 5 presents the corresponding event studies for the sync effect across all outcomes. Several features are worth noting. First, pre-trends are flat for all outcomes, supporting the parallel trends assumption. Second, the effects on lines of code, files, and commits are stable from roughly week 5 onward, consistent with the patterns in the main commit-level event studies. Third, the releases result is the least visually sharp: rather than a clear discontinuity at adoption, the effect builds gradually, which is consistent with the lag between writing code and shipping a release but also makes it harder to distinguish from a slow-moving pre-trend. We therefore interpret the releases result with caution.



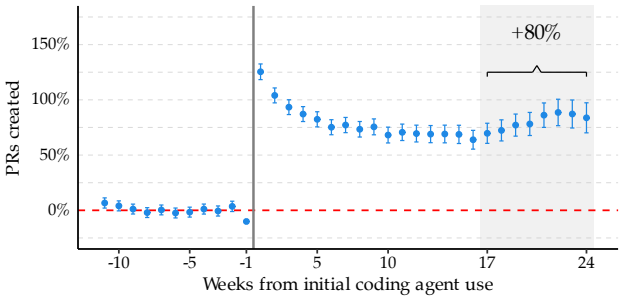
(a) Lines Changed



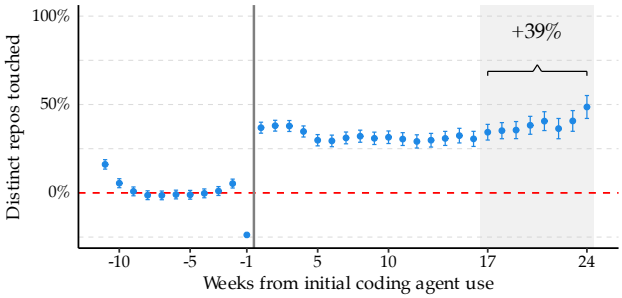
(b) Distinct Files



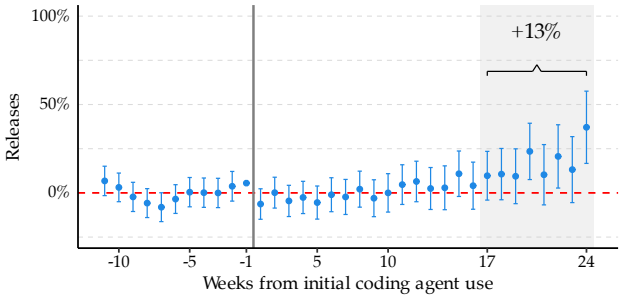
(c) Commits



(d) Pull Requests Created



(e) Distinct Repositories



(f) Releases

Figure 5: Sync Effect Across Production Layers

Note: 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 large transitory effects distort the scale.

6.4 Robustness

Our analysis relies on observational data, so we present several robustness checks that support a causal interpretation of our estimates.

6.4.1 Comparison to non-AI coding tools

We evaluate two additional coding tools in Figure 6 as placebo and bounding exercises.

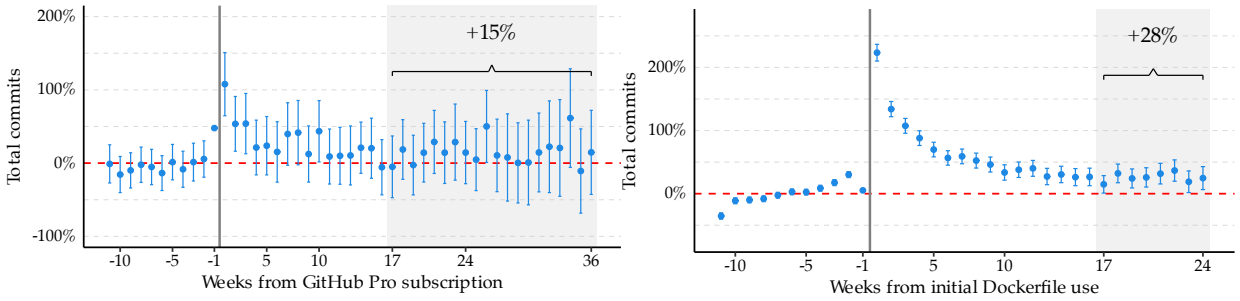
In the left panel, we study the adoption of GitHub Pro, a paid subscription that provides advanced collaboration and compute features but no AI capabilities. GitHub Pro is of interest because willingness to pay may signal periods of heightened engagement with the platform, potentially contaminating estimates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

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

As shown in Figure 6(b), Docker adoption is associated with a large short-run increase in activity, which may reflect either temporary activity bias or the resolution of a blocking technical constraint. This effect declines rapidly, however. While the productivity gains from coding agents stabilize at levels exceeding 100 percent, the Docker effect stabilizes at approximately 25 percent. 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 75% increase in productivity from the adoption of coding agents.

In Appendix Figures OA-11 and OA-12, 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.[12] However, we continue to find a large and

[12]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.



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

Figure 6: Productivity Effects of Other Coding Tools

Note: 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 6(a) compares users who subscribed to GitHub Pro in 2022 to a matched control group of users who authored a pull request and a commit a year prior on the same day. Panel 6(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 176% and 485% for Panels 6(a) and 6(b), respectively.

persistent effect from the adoption of all AI coding tools.

Table 6 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.

6.4.2 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 6). It is therefore reassuring that the treatment effects from coding agents remain stable over 24 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 9–16" and "Weeks 17–24".

6.4.3 Extensive vs Intensive Margin

As with any percentage treatment effect, our normalized estimates aggregate across both the extensive margin (whether coding tools increase the number of active developers) and the intensive margin (how much more code active developers write). Because we normalize by pre-period activity, our estimates weight sporadic coders heavily relative

Table 6: Productivity Effects Across Production Layers for Other Coding Tools

	Outcomes (Weeks 17–24, %)					
	Lines Chg	Dist Files	Commits	PRs	Dist Repos	Releases
GitHub Pro	174.7	30.2	15.0	−35.4	2.0	22.8
	(82.6)	(20.4)	(11.7)	(15.4)	(6.4)	(23.6)
Docker	229.8	87.1	28.3	28.9	15.4	−6.9
	(35.4)	(7.2)	(4.7)	(4.8)	(1.8)	(8.7)

Notes: This table reports event-study coefficients (1% winsorized, weeks 17–24 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.

to highly active ones. In the presence of heterogeneous level effects—consistent with the patterns in Figure 4—the aggregate effect on total commits would be smaller than the average percentage effect we report.

6.4.4 External validity: public vs. private repositories

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

7 Aggregate Software Output

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), noting the puzzling disconnect between rapid advances in information technology and sluggish aggregate productivity growth. Decades later, despite the transformative potential of AI, a similar concern has emerged: while individual studies

consistently document large task-level productivity gains (e.g., Peng et al., 2023; Noy and Zhang, 2023; Brynjolfsson et al., 2025; Cui et al., 2025), it remains unclear whether these gains are visible in aggregate output. Indeed, recent work has emphasized the potential disconnect between task-level AI capabilities and economy-wide productivity growth (Acemoglu, 2025; Jones, 2026).

In the context of software development, aggregate productivity corresponds to the output of the applications industry—the total quantity and quality of software products available to end users. Unlike aggregate productivity in the broader economy, which is notoriously difficult to measure and to attribute to any particular technology, software output can be observed relatively directly through application marketplaces. These platforms record when new applications are published, when existing applications are updated, and what features each version contains. This provides a concrete empirical analog to the abstract concept of aggregate productivity: if AI coding tools make developers more productive, we should eventually observe more software being produced, updated, or improved.

We examine these questions in two parts. First, we study aggregate coding activity on GitHub, the dominant platform for collaborative software development. This analysis provides a direct aggregate counterpart to our developer-level event studies: whereas the event studies estimate the effect of AI tool adoption on individual developers' commits, pull requests, and releases, the aggregate analysis asks whether these effects are visible in platform-wide totals. Second, we turn to the output and quality of software on major distribution marketplaces, examining whether the productivity gains we document translate into more applications being created, updated more frequently, and consumed by end users.

7.1 Software Marketplaces: Data and Measurement

To assess whether AI-driven productivity gains translate into increases in actual software output, we examine trends across three major software distribution marketplaces: the VS Code Extension Marketplace, SourceForge, and the Apple App Store.

These three marketplaces span a broad cross-section of the software ecosystem, differing in their target users, platforms, and the types of software they distribute. SourceForge is one of the oldest software distribution platforms and skews heavily toward legacy software, desktop applications, and utilities for Linux and Windows. The VS Code Extension Marketplace caters to software developers, distributing IDE extensions that enhance coding workflows—and whose developers are likely among the earliest and most intensive adopters of AI coding tools. The Apple App Store targets mobile consumers on iOS, cov-

ering everything from games and social media to productivity and finance applications. By examining trends across all three, we can assess whether aggregate productivity effects are broad-based or confined to particular segments of the software economy.

Because these datasets originate from different marketplaces, the variables we observe are not uniform across platforms. In all three marketplaces, we observe the entry of new developers and applications. In the VS Code Extension Marketplace and SourceForge, we observe version histories and release dates, allowing us to measure update frequency. For the VS Code Extension Marketplace, we additionally observe the new features released with each update. SourceForge provides download counts. This heterogeneity in observables means that different marketplaces contribute to different parts of our analysis.

If AI coding tools increase developer productivity, we would expect this to manifest through three channels. First, on the *extensive margin*, lower costs of software creation may lead to more new applications being developed and published. Second, on the *intensive margin*, existing applications may ship more features as developers become more productive. Third, also on the intensive margin, existing applications may be updated more frequently, as the time required to implement changes and fixes declines.

7.2 Aggregate Coding Activity on GitHub

Figures 7(a)–7(d) present monthly aggregate activity on GitHub from 2019 through early 2026. We plot four platform-wide metrics: the number of new repositories created, the number of active repositories (those receiving at least one push), the number of new user accounts, and the number of pull requests merged. All four panels display a broadly linear trend from 2019 through 2024, reflecting steady organic growth in the platform's user base and activity. Around early 2025, this trend breaks visibly: all four metrics exhibit a clear increase in growth rate relative to the pre-2025 trend.

This break coincides with the period in which sync and async coding agents became widely available, and the magnitude of the acceleration is broadly in line with the individual-level estimates of a 30–100% increase in commits. However, the aggregate patterns are consistent with the developer-level effects only if aggregate adoption of AI coding tools is sufficiently large. To assess this, Figure 7(e) plots the share of commits on public repositories that are signed or authored by just one agentic coding tool—Claude Code. The share is rapidly rising, its timeline aligns with the acceleration in overall GitHub activity visible in Figures 7(a)–7(d), and, at above 5%, the fraction of commits attributable to this single tool is already substantial. 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.

7.3 Extensive Margin: New Applications

We now examine whether the productivity gains documented above translate into more software reaching end users, by studying the number of new applications published on distribution marketplaces.

Figure 8 plots the number of newly created applications per month for three of the marketplaces. The VS Code Extension Marketplace (Panel 8(a)) shows a gradual increase from approximately 500 new extensions per month in 2019 to roughly 1,000 per month by 2024, followed by a sharp acceleration beginning around January 2025, reaching approximately 3,000 per month by early 2026. By contrast, SourceForge (Panel 8(b)) shows no discernible upward trend: new projects fluctuate noisily between roughly 250 and 650 per month throughout the sample, with if anything a slight decline over time and no acceleration following the adoption of AI tools. This null result is informative, as SourceForge caters primarily to legacy desktop software—a segment whose developers may be less likely to adopt AI coding tools. The Apple App Store (Panel 8(c)) displays steady growth from approximately 8,000 new apps per month in 2019 to around 20,000 by early 2025, followed by a slight acceleration in the second half of 2025, with new apps roughly doubling to over 40,000 per month by early 2026.

Overall, the extensive-margin evidence is consistent with—but more heterogeneous than—the GitHub aggregate trends. The VS Code Extension Marketplace and the Apple App Store both display trend breaks around 2025, while SourceForge shows no acceleration. This heterogeneity suggests that the translation from developer-level productivity gains to aggregate software output depends on the type of software, the characteristics of the developer population, and the barriers to publication on each platform.

7.4 Intensive Margin: Update Frequency and New Features

We now turn to the intensive margin, asking whether existing applications are being updated more frequently or shipping more features. A simple year-over-year comparison—contrasting update behavior in 2025 with that in 2024—could be misleading if update frequency was already trending upward before the widespread adoption of AI coding tools. To address this, we estimate a specification that controls for the pre-existing trend and identifies whether 2025 represents an *acceleration* beyond what a linear trend would predict.

Specifically, we pool all updates from January 2023 through March 2026 and estimate:

$$y_{ijt} = \alpha + \tau\, s_t + \eta\, \mathbf{1}\{t \in \text{H2}\} + \beta_1\, \mathbf{1}\{t \in \text{H1 2025}\} + \beta_2\, \mathbf{1}\{t \in \text{H2 2025}\} + \gamma_j + \delta(\text{age}_{it}) + \varepsilon_{ijt}, \tag{11}$$

Table 7: Aggregate Effects Across Software Marketplaces

	VS Code Extensions	Source-Forge	Apple App Store
Panel A: Extensive Margin			
New Apps (β_1: H1 2025)	0.209	-0.184	0.145
	(0.103)	(0.102)	(0.091)
New Apps (β_2: H2 2025)	0.446	0.107	0.304
	(0.104)	(0.090)	(0.076)
Panel B: Intensive Margin			
Update Frequency (β_1: H1 2025)	0.027	0.047	—
	(0.015)	(0.024)	
Update Frequency (β_2: H2 2025)	0.209	0.336	—
	(0.019)	(0.031)	
New Features (β_1: H1 2025)	-0.003	—	—
	(0.042)		
New Features (β_2: H2 2025)	0.373	—	—
	(0.050)		
Panel C: Consumption			
Downloads/Ratings (β_1: H1 2025)	—	-0.091	—
		(0.093)	
Downloads/Ratings (β_2: H2 2025)	—	-0.068	—
		(0.113)	
Mean Update Freq. (2024)	0.036	0.068	—
Mean New Features (2024)	0.006	—	—
# of Applications	51,276	14,014	894,389

Notes: This table reports estimates of β_1 and β_2 from the aggregate specification across three software marketplaces. All coefficients can be interpreted as approximate percentage changes relative to what the linear trend would predict (e.g., 0.209 corresponds to a 20.9% increase). In Panel A (extensive margin), the dependent variable is the log monthly count of new applications; standard errors are Newey-West HAC. In Panel B (intensive margin), the dependent variable is a binary indicator for whether an update or new feature was released, renormalized by the 2024 mean frequency so that coefficients are interpretable as percentage changes; all specifications include update-number fixed effects and app age controls. In Panel C (consumption), we estimate a Poisson specification so that coefficients are again interpretable as percentage changes. Dashes indicate outcomes not available for a given marketplace.

where y_{ijt} is the outcome for application i's j-th update occurring at date t (either the time elapsed since the previous update or the number of new features released), and s_t is a linear half-year trend indexing the six half-year periods in our sample (H1 2023 = 0, H2 2023 = 1, . . . , H2 2025 = 5). The trend τ absorbs the secular rate of change in update behavior across the three years of our sample, and $\mathbf{1}\{t \in \text{H2}\}$ absorbs seasonal differences between the first and second halves of the calendar year. Given these controls, β_1 and β_2 have a difference-in-differences interpretation: β_1 estimates the change in update behavior from H1 2024 to H1 2025, net of the change from H1 2023 to H1 2024, and β_2 is defined analogously for the second half of the year. That is, β_1 and β_2 measure whether update behavior *accelerated* in 2025 beyond the pre-existing trend.[13] Finally, γ_j are update-number fixed effects and $\delta(\text{age}_{it})$ controls for the age of the application at the time of the update.

An important caveat of this specification is that the linear trend assumption may not hold. If the true counterfactual trend were instead convex (e.g., exponential), the linear specification would attribute the curvature to β_1, yielding a spuriously positive estimate. In future iterations of this paper, we will present robustness checks using more flexible trend specifications (e.g., quadratic trends) and provide visual evidence on pre-period trend linearity in the appendix. Our controls also already address an additional confound: younger applications tend to have higher update frequency and more feature releases than mature ones, reflecting the rapid iteration typical of early-stage development. If the composition of active applications shifts toward younger apps over time—for instance, because AI tools lower the barrier to entry, as suggested by our extensive-margin results—a naïve comparison could spuriously attribute this compositional shift to higher per-app productivity. Controlling for app age and update number ensures that we compare applications at the same stage of their lifecycle.

Table 7 reports the results. As a sanity check, Panel A applies the specification to the extensive margin—new application creation—where we can compare the regression estimates to the raw trends in Figure 8. We find strong positive effects on new apps for both the VS Code Extension Marketplace and the Apple App Store, consistent with the visual evidence discussed above. Turning to the intensive margin in Panel B, we find that update frequency increases in the second half of 2025 for VS Code extensions, and that the number of new features per update also rises. These results suggest that AI tools may be accelerating not only the creation of new software but also the pace at which

[13] To see why, consider the H1 periods: the linear trend implies $E[y \mid \text{H1 2023}] = \alpha$, $E[y \mid \text{H1 2024}] = \alpha + 2\tau$, and $E[y \mid \text{H1 2025}] = \alpha + 4\tau + \beta_1$. The year-over-year change from H1 2023 to H1 2024 is 2τ, while the change from H1 2024 to H1 2025 is $2\tau + \beta_1$. Hence β_1 is exactly the double difference: the 2025–2024 change minus the 2024–2023 change. The identifying assumption is that, absent AI tools, the pre-existing rate of change would have continued linearly into 2025.

existing applications are improved. We note, however, that the intensive-margin estimates are more speculative than our developer-level event studies: they rely on the assumption that the pre-existing linear trend would have continued into 2025, and violations of this assumption could bias the estimates.

7.5 Consumption

The extensive-margin results above document an increase in the number of new applications, but this does not necessarily imply an increase in consumer welfare. The marginal applications created with the help of AI tools may be small, experimental, or unused—incremental projects that are published but never adopted. To assess whether the supply-side gains documented above are reaching end users, we examine whether applications released in 2025 are consumed more or less than applications released in earlier years at the same point in their lifecycle.

We estimate the same specification as in equation (11), but replace the outcome with a measure of consumption: download counts on SourceForge, which serve as a proxy for user adoption. Because download counts are extremely right-skewed, we estimate the specification using Poisson pseudo-maximum likelihood (Silva and Tenreyro, 2006). As before, the update-number fixed effects and app age controls ensure that we compare applications at the same lifecycle stage, and the linear trend and 2025 indicators retain their difference-in-differences interpretation. A positive β would indicate that 2025-vintage applications attract more consumption than their 2024 counterparts did at the same age, suggesting that the new supply is reaching users. A zero or negative β would suggest dilution—more applications competing for the same pool of users, with no net increase in per-app consumption.

As shown in Panel C of Table 7, we find no significant effect on downloads for SourceForge. This is perhaps unsurprising given that we also found no effect on new application creation on that platform. However, the null result on consumption does cast some doubt on the broader interpretation of the intensive-margin results: if more software is being produced and updated, it is not yet clear that users are consuming more of it. That said, it is entirely possible for AI tools to improve the quality or frequency of software updates without increasing consumption, if consumers have limited time or attention to adopt additional applications. These results are speculative, and in future iterations of this paper we plan to incorporate data on ratings from the Apple App Store as a proxy for consumption on a platform where we do observe extensive-margin effects.

7.6 Discussion

The aggregate evidence complements our developer-level findings. Coding activity on GitHub has accelerated sharply since early 2025, consistent with the individual-level productivity gains we document. The translation into shipped software is more uneven: the VS Code Extension Marketplace shows a dramatic increase in new applications, while other platforms show weaker or no acceleration—consistent with the model's prediction that productivity gains attenuate as output moves through human-bottlenecked layers of the production hierarchy. Taken together, the aggregate patterns suggest that AI coding tools are beginning to affect not just how much code developers write, but how much software reaches end users, though the magnitude and breadth of these effects remain early-stage.

An important caveat is that the increases documented above do not necessarily reflect productivity gains. An alternative explanation is that AI has generated new demand for software—particularly AI-related applications—and the market has responded to this demand expansion regardless of any change in developer productivity. Disentangling demand-driven entry from productivity-driven entry is inherently difficult; in future iterations of this paper, we plan to examine the composition of new applications to determine what share are directly AI-related and to restrict attention to applications released before 2025, whose update behavior cannot be attributed to demand expansion.

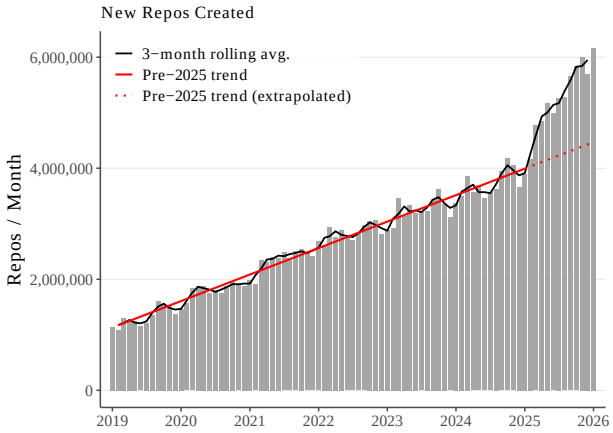
8 Conclusion

A common refrain from employees of AI labs is that "large language models today are the worst they will ever be." As benchmark performance improves, do these gains translate into higher real-world productivity? This is the question we set out to answer in this paper. We find that successive generations of AI coding tools produce increasingly large productivity effects, with sync agents roughly doubling developer output relative to autocomplete alone. However, these gains attenuate sharply across stages of the production hierarchy: effects are large for lines of code and commits but much smaller for pull requests and releases. At the aggregate level, coding activity on GitHub has accelerated since early 2025, and new application creation has increased on some software marketplaces, though the effects are heterogeneous across platforms.

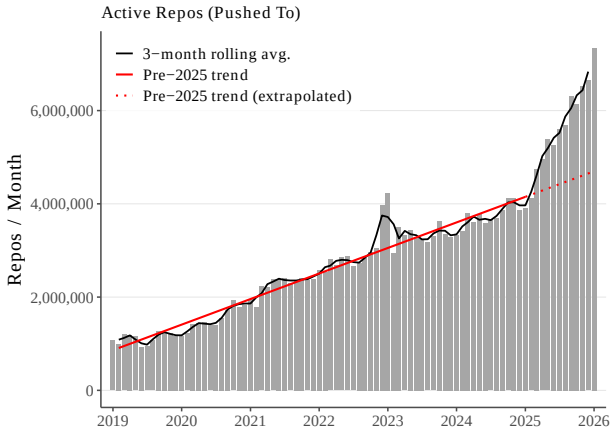
Two key limitations of our study are our reliance on observational data and our focus on public repositories. Estimating the effects of technology adoption using observational data presents well-known challenges, and activity on public repositories may not be representative of developers' overall coding behavior. We take these concerns seriously and present a series of robustness checks—including placebo tests with non-AI tools, stable

effects over 24 weeks, and a comparison to internal Microsoft data on private-repository usage—that support a causal interpretation of our estimates.

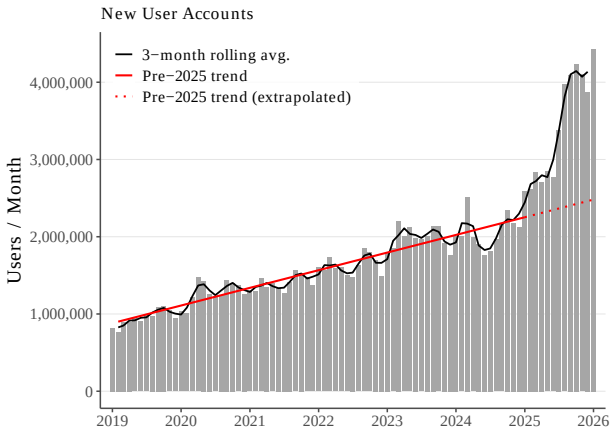
Our findings have implications beyond software development. The production model we develop shows that when stages of production are complementary, automating one stage has bounded effects on final output—a vertical analog of the "weak links" logic in the macroeconomic growth literature. In software, the binding constraint appears to be shifting from writing code to reviewing and integrating it. Whether future generations of AI tools can ease this constraint—by producing higher-quality code that requires less human review, or by automating the review and integration process itself—will determine whether the large task-level productivity gains we document ultimately translate into commensurate increases in shipped software. The industry is changing rapidly, and we intend to update this paper as additional data on the adoption and productivity effects of AI coding tools become available.



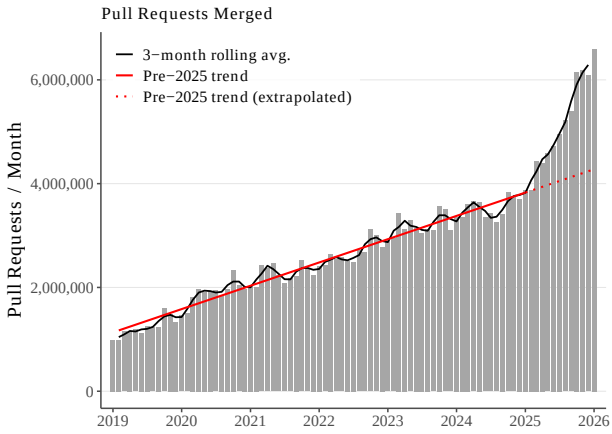
(a) New Repos Created



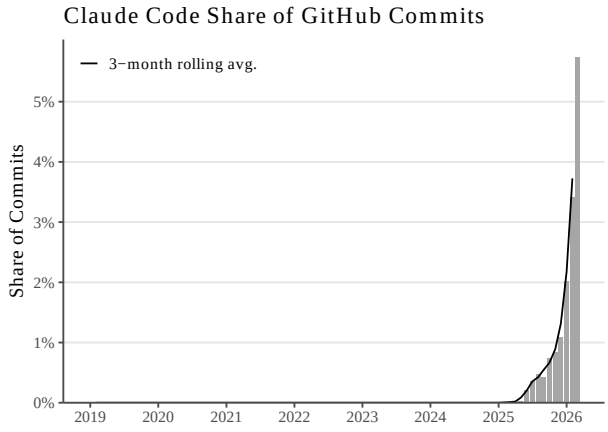
(b) Active Repos (Pushed To)



(c) New User Accounts



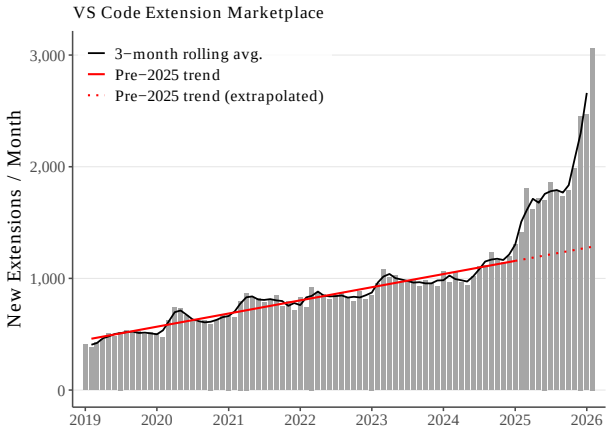
(d) Pull Requests Merged



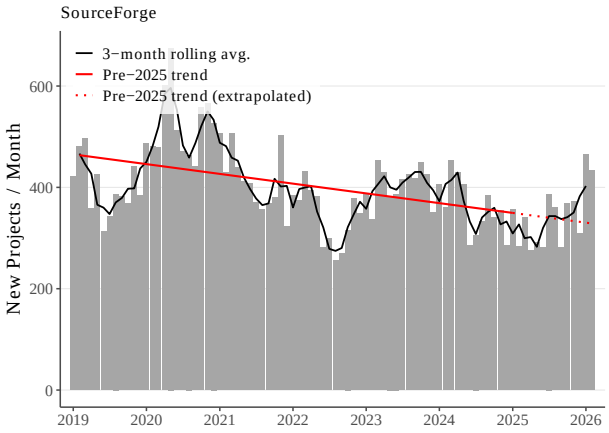
(e) Claude Code Commit Share

Figure 7: Aggregate Coding Activity on GitHub

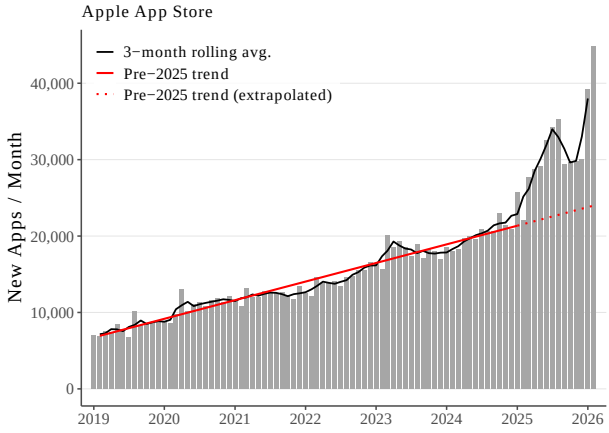
Note: This figure plots monthly aggregate activity on public GitHub repositories from 2019 through early 2026, using data from the GitHub Search API. Bars show monthly counts; the solid line is a rolling average. New repos and active repos count non-fork repositories created and pushed to, respectively. Pull requests merged and new user accounts count all merged PRs and individual (non-organization) accounts created on public repositories. 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.



(a) VS Code Extensions



(b) SourceForge



(c) Apple App Store

Figure 8: New Applications Across Software Marketplaces

Note: This figure plots the number of newly created applications per month across three software distribution marketplaces. VS Code extensions are obtained from the VS Code Marketplace API; we observe creation dates for all extensions. SourceForge project histories are obtained from the SourceForge website. Apple App Store data are from a cross-section of applications pulled directly from Apple, with creation dates used to construct a time series of new entries. Bars show monthly counts; the solid line is a rolling average.

References

Acemoglu, D. (2025). The simple macroeconomics of ai. *Economic Policy 40*(121), 13–58.

Acemoglu, D. and P. Restrepo (2019, May). 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.

Agrawal, A., J. Gans, and A. Goldfarb (2018). *Prediction Machines: The Simple Economics of Artificial Intelligence*. Harvard Business Review Press.

Becker, J., N. Rush, E. Barnes, and D. Rein (2025). Measuring the impact of early-2025 ai on experienced open-source developer productivity.

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, January). The productivity J-curve: How intangibles complement general purpose technologies. *American Economic Journal: Macroeconomics 13*(1), 333–372.

Campero, A., M. Vaccaro, J. Song, H. Wen, A. Almaatouq, and T. W. Malone (2022). A test for evaluating performance in human-computer systems.

Cui, Z., M. Demirer, S. Jaffe, L. Musolff, S. Peng, and T. Salz (2025, August). The effects of generative AI on high-skilled work: Evidence from three field experiments with software developers. Technical report, SSRN. Available at SSRN.

Dell'Acqua, F., E. McFowland, E. R. Mollick, H. Lifshitz-Assaf, K. Kellogg, S. Rajendran, L. Krayer, F. Candelon, and K. R. Lakhani (2023). Navigating the jagged technological frontier: Field experimental evidence of the effects of ai on knowledge worker productivity and quality. *Harvard Business School Technology & Operations Mgt. Unit Working Paper* (24-013).

Eloundou, T., S. Manning, P. Mishkin, and D. Rock (2024). Gpts are gpts: Labor market impact potential of llms. *Science 384*(6702), 1306–1308.

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, January). A.I. and our economic future. Working Paper 34779, National Bureau of Economic Research.

Jones, C. I. and C. Tonetti (2026, January). Past automation and future A.I.: How weak links tame the growth explosion. Unpublished manuscript.

Kremer, M. (1993). The O-ring theory of economic development. *The Quarterly Journal of Economics 108*(3), 551–575.

Noy, S. and W. Zhang (2023). Experimental evidence on the productivity effects of generative artificial intelligence. *Science 381*(6654), 187–192.

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*.

Sarkar, S. K. (2025, November). Ai agents, productivity, and higher-order thinking: Early evidence from software development. Technical report, SSRN. Available at SSRN.

Silva, J. M. C. S. and S. Tenreyro (2006). The log of gravity. *The Review of Economics and Statistics 88*(4), 641–658.

Solow, R. M. (1987, July 12). We'd better watch out. *New York Times Book Review*, 36.

Vaithilingam, P., T. Zhang, and E. L. Glassman (2022). Expectation vs. experience: Evaluating the usability of code generation tools powered by large language models. In *CHI Conference on Human Factors in Computing Systems Extended Abstracts*, pp. 1–7.

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 Model (Pass-Through in a Single CES Layer) . . . . . **OA - 2**
A.1 Setup OA - 2
A.2 Exact Pass-Through Formula OA - 2
A.3 Comparative Statics OA - 3
A.4 Proof: Shape of Attenuation Across Layers OA - 4
A.5 Special Cases OA - 5
A.6 Ordering by σ OA - 6
A.7 Full Automation: When Upstream Input Becomes Free OA - 6
A.8 Numerical Illustration OA - 7
A.9 Chaining Across Layers OA - 8

B Data and Estimation . . . . . **OA - 9**
B.1 Outcome Variables OA - 9
B.2 Data Collection & Sample Construction OA - 10
B.3 Estimation Details OA - 16

C Descriptive Facts about Usage of AI Coding Tools . . . . . **OA - 23**
C.1 Adoption OA - 23
C.2 Stickiness of Usage of Coding Tools OA - 24

D Learning . . . . . **OA - 26**

E Additional Figures . . . . . **OA - 26**

A Model (Pass-Through in a Single CES Layer)

This appendix derives the exact effect of a productivity shock on output in a single CES production layer. The result depends only on the elasticity of substitution σ and the revenue share of the input being shocked, θ—not on the structural share parameter α.

A.1 Setup

Consider a single layer of production that combines 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, $x = y_{s-1}$ is the output from the layer below and $e = e_s$ is effective effort at the current layer.

The revenue share of upstream input is:

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

Under competitive markets, θ equals the cost share of upstream input, which is in principle observable from data.

A.2 Exact Pass-Through Formula

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

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

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

Proposition 4 (Exact pass-through)**.** *A shock that multiplies upstream input by $A > 1$ changes output by:*

$$\boxed{\frac{\tilde{y}}{y} = [\theta\, A^{\rho} + (1-\theta)]^{1/\rho},} \tag{15}$$

where θ is the initial revenue share of upstream input and $\rho = (\sigma - 1)/\sigma$.

Proof. From the ratio $\tilde{y}/y$:

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

□

Key feature. The pass-through formula depends on only two objects: (i) θ, the revenue share of upstream input (observable from cost data), and (ii) σ, the elasticity of substitution (to be estimated or calibrated). The structural share parameter α does not appear separately—it enters only through θ.

A.3 Comparative Statics

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

1. Shock size A. The pass-through is increasing in A:

$$\frac{\partial G}{\partial A} = \theta\, A^{\rho-1}\, [\theta\, A^{\rho} + (1-\theta)]^{\frac{1}{\rho}-1} > 0. \tag{16}$$

The economically relevant object is the elasticity of pass-through:

$$\eta(A) \equiv \frac{\partial \ln G}{\partial \ln A} = \frac{\theta\, A^{\rho}}{\theta\, A^{\rho} + (1-\theta)}, \tag{17}$$

with derivative $\partial\eta/\partial A = \theta(1-\theta)\rho\, A^{\rho-1}/[\theta A^{\rho} + (1-\theta)]^2$, whose sign is the sign of $\rho = (\sigma-1)/\sigma$. With complements ($\sigma < 1$), η is decreasing in A: each additional percentage increase in the shock translates into a *smaller* percentage gain in output. With substitutes ($\sigma > 1$), η is increasing: each additional percentage increase translates into a *larger* gain. Under Cobb–Douglas, $\eta(A) = \theta$ for all A.

2. Revenue share θ. The pass-through is increasing in θ:

$$\frac{\partial G}{\partial \theta} = \frac{1}{\rho}\, [\theta\, A^{\rho} + (1-\theta)]^{\frac{1}{\rho}-1}\, (A^{\rho} - 1). \tag{18}$$

Since $A > 1$: when $\rho > 0$ ($\sigma > 1$), $A^{\rho} > 1$ and the prefactor $1/\rho > 0$, 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, $G = A^{\theta}$ and $\partial G/\partial\theta = A^{\theta} \ln A > 0$.

Interpretation: layers where upstream input accounts for a larger share of costs transmit upstream shocks more fully. In our setting, lower layers of the hierarchy—where code is a

larger share of the "input"—are more responsive to AI-driven increases in code generation.

3. Elasticity of substitution σ. The pass-through is increasing in σ:

$$\frac{\partial G}{\partial \sigma} > 0 \qquad \text{for all } A > 1,\ \theta \in (0,1). \tag{19}$$

This is the content of Proposition 5 below: G is a power mean of A and 1, and power means are increasing in their order. The economic intuition is that higher substitutability allows the layer to take greater advantage of abundant upstream input, rather than being constrained by the scarce input (effort).

4. Share parameter α. The pass-through formula (15) is expressed in terms of the revenue share θ, which absorbs α. To isolate the effect of α, return to the primitive expression. From (13), $\theta = \alpha x^\rho / [\alpha x^\rho + (1-\alpha) e^\rho]$, so:

$$\frac{\partial \theta}{\partial \alpha} = \frac{x^\rho\, e^\rho}{[\alpha\, x^\rho + (1-\alpha)\, e^\rho]^2} > 0. \tag{20}$$

Since $\partial G/\partial \theta > 0$ and $\partial \theta/\partial \alpha > 0$, the chain rule gives $\partial G/\partial \alpha > 0$: a higher share parameter on upstream input raises the pass-through. Intuitively, when the production technology places more weight on upstream input (α higher), the layer is more responsive to changes in that input.

Summary of comparative statics.

Parameter	Effect on pass-through G	Intuition
A (shock size)	$\partial G/\partial A > 0$	Larger shocks raise output
	Elasticity η decreasing if $\sigma < 1$	. . . with diminishing proportional gains under comple
	Elasticity η increasing if $\sigma > 1$	. . . with increasing proportional gains under substitut
θ (upstream share)	$\partial G/\partial \theta > 0$	Layers more reliant on upstream input transmit more
α (share parameter)	$\partial G/\partial \alpha > 0$	Higher weight on upstream input $\Rightarrow$ more transmission
σ (substitutability)	$\partial G/\partial \sigma > 0$	More substitutable inputs $\Rightarrow$ less attenuation

A.4 Proof: Shape of Attenuation Across Layers

We prove Proposition 3(1). Consider the symmetric case with common σ and θ at every layer. Let $L_s = \ln G_s$ and $R_s = L_{s+1}/L_s$. Since $\ln L_s = \ln \ln G_s$, strict convexity (concavity) of $\{\ln \ln G_s\}$ in s is equivalent to R_s being strictly increasing (decreasing) in s. We show the latter.

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 $f(1) = 1$, the fundamental theorem of calculus gives $\ln f(x) = \int_0^{\ln x} \eta(e^u)\, du$, 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 $\sigma > 1$, the integrand $\eta(e^u)$ is strictly increasing in u, so the running mean lies strictly below the endpoint: $\bar{\eta}(x) < \eta(x)$, giving $\bar{\eta}'(x) > 0$. 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 (each layer attenuates):

- $\sigma < 1$: $\bar{\eta}$ decreasing, G_s decreasing $\Rightarrow$ R_s increasing $\Rightarrow$ $\{\ln \ln G_s\}$ convex.
- $\sigma = 1$: $\bar{\eta} \equiv \theta \Rightarrow R_s = \theta \Rightarrow \{\ln \ln G_s\}$ linear.
- $\sigma > 1$: $\bar{\eta}$ increasing, G_s decreasing $\Rightarrow$ R_s decreasing $\Rightarrow$ $\{\ln \ln G_s\}$ concave. □

A.5 Special Cases

Cobb–Douglas ($\sigma = 1$, $\rho \to 0$). Taking the limit:

$$\frac{\tilde{y}}{y} = A^\theta. \tag{21}$$

The pass-through is A^θ, independent of the scale of the shock.

Leontief ($\sigma \to 0$, $\rho \to -\infty$).

$$\frac{\tilde{y}}{y} \to 1 \qquad \text{whenever } x > e \text{ (upstream input is abundant).} \tag{22}$$

If upstream input already exceeds effective effort, increasing it further has no effect.

Perfect substitutes ($\sigma \to \infty$, $\rho \to 1$).

$$\frac{\tilde{y}}{y} = \theta\, A + (1 - \theta). \tag{23}$$

A.6 Ordering by σ

Proposition 5 (Ordering by σ)**.** *For any $A > 1$ and $\theta \in (0, 1)$, the pass-through is strictly increasing in σ:*

$$\underbrace{1}_{\textit{Leontief}} \;<\; \underbrace{[\theta A^{\rho} + (1-\theta)]^{1/\rho}}_{\textit{CES, } \sigma<1} \;<\; \underbrace{A^{\theta}}_{\textit{Cobb–Douglas}} \;<\; \underbrace{[\theta A^{\rho} + (1-\theta)]^{1/\rho}}_{\textit{CES, } \sigma>1} \;<\; \underbrace{\theta A + (1-\theta)}_{\textit{Perfect subs.}}$$

The more substitutable the inputs, the larger the output gain from a given upstream shock.

Proof. The expression $[\theta A^{\rho} + (1-\theta)]^{1/\rho}$ is a power mean of A and 1 with weights θ and $1-\theta$. Power means are increasing in their order parameter ρ, and $\rho = (\sigma-1)/\sigma$ is increasing in σ. □

A.7 Full Automation: When Upstream Input Becomes Free

The sharpest way to see the role of σ is to consider the limiting case of *full automation*: AI makes upstream input essentially free, so that its quantity can grow without bound ($x \to \infty$). Whether this translates into large or small output gains depends *entirely* on whether upstream input and effective effort are complements or substitutes.

Taking $A \to \infty$ in the pass-through formula (15):

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

$$\left.\frac{\tilde{y}}{y}\right|_{A\to\infty} = (1-\theta)^{\frac{\sigma}{\sigma-1}}. \tag{24}$$

  Output converges to a *finite* limit. Even with infinite upstream input, the gain is bounded. No matter how much code AI generates—even an infinite amount—the output at the next layer cannot exceed a ceiling determined by θ and σ. The scarce input (human effort) is the binding constraint, and no amount of the abundant input can compensate for it.
- **Cobb–Douglas** ($\sigma = 1$): $\tilde{y}/y = A^{\theta} \to \infty$. Output grows without bound, but slowly (as a power of A). This is the knife-edge case where the bound just disappears.
- **Substitutes** ($\sigma > 1$, $\rho > 0$): $\tilde{y}/y \to \infty$. Output grows without bound and faster

than under Cobb–Douglas. Abundant upstream input can effectively replace human effort, so making code free translates into unbounded output gains.

This is the central distinction. With complements, full automation of one input has bounded effects—the other input becomes the bottleneck. With substitutes, full automation of one input has unbounded effects—the automated input can substitute for the other. The question of whether AI-generated code and human review effort are complements or substitutes is therefore not merely quantitative; it determines whether the *qualitative* nature of the gains is bounded or unbounded.

Connection to Jones (2026). The bound (24) generalizes the $\frac{1}{1-s}$ result in Jones (2026). In Jones's example with $\sigma = 1/2$ ($\rho = -1$), the exponent is $\sigma/(\sigma-1) = -1$, so the bound becomes $(1-\theta)^{-1} = \frac{1}{1-\theta}$—exactly his formula, with the spending share θ playing the role of s. For general $\sigma < 1$, the bound is:

σ	$\sigma/(\sigma-1)$	Bound $(1-\theta)^{\sigma/(\sigma-1)}$ with $\theta = 0.5$	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%. With $\sigma = 0.5$ (Jones's benchmark), the maximum gain is a doubling. The bound increases sharply as σ approaches 1, and at the Cobb–Douglas knife-edge ($\sigma = 1$) it disappears entirely. The empirical question of whether σ is above or below 1 therefore has first-order consequences for the long-run impact of AI on software production.

A.8 Numerical Illustration

Consider a shock $A = 7.5$ (motivated by the empirical finding that sync agents increase lines of code by roughly 650%), with $\theta = 0.5$:

Elasticity of substitution σ	$\rho = (\sigma-1)/\sigma$	Output gain $\tilde{y}/y$
0 (Leontief)	$-\infty$	1.00 (no effect)
0.25	-3	1.26
0.5	-1	1.76
0.75	$-1/3$	2.32
1.0 (Cobb–Douglas)	0	2.74
1.5	$1/3$	3.23
2.0	$1/2$	3.49
∞ (perfect subs.)	1	4.25

A.9 Chaining Across Layers

For two layers in sequence, the output gain at layer 1 becomes the input shock for layer 2. If the shock at layer 1 gives $\tilde{y}_1/y_1 = G_1$, then the effect at layer 2 is:

$$\frac{\tilde{y}_2}{y_2} = \left[\theta_2 \, G_1^{\rho_2} + (1 - \theta_2)\right]^{1/\rho_2}. \tag{25}$$

In the symmetric case ($\sigma_s = \sigma$, $\theta_s = \theta$ for all s), this can be computed iteratively: define $G_0 = A$ and $G_s = [\theta \, G_{s-1}^{\rho} + (1 - \theta)]^{1/\rho}$ for $s = 1, \ldots, S - 1$.

Numerical example: four layers of pass-through. Starting from a 7.5× shock at layer 1, with $\theta = 0.5$ at every layer:

	Output gain after passing through ℓ layers			
σ	$\ell = 1$	$\ell = 2$	$\ell = 3$	$\ell = 4$
0.25	1.26	1.10	1.05	1.02
0.50	1.76	1.28	1.12	1.06
0.75	2.32	1.48	1.21	1.10
1.00 (CD)	2.74	1.65	1.29	1.13
1.50	3.23	1.90	1.40	1.19

After four layers of pass-through (from code to apps), a 7.5× code increase yields:

- A **2% increase** in apps under $\sigma = 0.25$,
- A **6% increase** under $\sigma = 0.5$,
- A **13% increase** under Cobb–Douglas ($\sigma = 1$),
- A **19% increase** under $\sigma = 1.5$.

Calibration. The empirical finding that a ~750% increase in lines of code corresponds to a ~10–15% increase in apps is consistent with σ between approximately 0.75 and 1.0 when $\theta = 0.5$. Under $\sigma = 0.75$, the model predicts a 10% increase; under Cobb–Douglas ($\sigma = 1$), a 13% increase. This range aligns with the estimates in the macroeconomic literature: Jones and Tonetti (2026) use $\sigma = 0.5$ as their baseline, and Chirinko (2008) and Raval (2019) report estimates of 0.3–0.7. If θ were higher (e.g., $\theta = 0.7$), the data would be consistent with lower values of σ.

B Data and Estimation

B.1 Outcome Variables

Our primary outcome measures are based on commits and pull requests, the fundamental units of code contribution on GitHub.

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).

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.

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.

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.

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.

Releases Repositories on GitHub can be 'released' by their owners, which typically involves tagging a commit as a release, uploading some related binaries, and posting a release note and/or changelog. We measure the number of releases created by each developer in each week. This allows us to get a sense to what extent developers are shipping code to their users.

B.2 Data Collection & Sample Construction

B.2.1 Overview

This appendix describes the data collection and construction procedures for all seven coding tools examined in this study. We categorize the tools into two groups based on their data sources and user identification strategies:

1. **AI Coding Agents with Direct Usage Signals**: GitHub Async Agent, Codex Async Agent, Claude Code Sync Agent, and Dockerfile. For these tools, we identify users through observable actions in the GitHub ecosystem—specifically, pull requests or file commits that directly indicate tool usage.

2. **AI Coding Tools with Subscription Data**: 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.

B.2.2 Coding Tool Usage Data

GitHub Async Agent 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[14].

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.

[14] https://docs.github.com/en/copilot/concepts/agents/coding-agent/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. See Tables OA-1 and OA-2 for the user-source distributions, and Figure OA-1 for the distribution of event dates.

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

User source	Count	Percent (%)
coauthor	451,936	65.08
requested reviewer	189,946	27.35
assignee	50,950	7.34
Missing	1,575	0.23
Total	694,407	100.00

Table OA-2: User-Source Distribution for Identified Copilot Agent Users

User source	Count	Percent (%)
coauthor	64,350	58.64
requested reviewer	38,842	35.39
assignee	6,554	5.97
Total	109,746	100.00

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[15]) or after the data collection date.

As of 2025-12-22, we retrieved 2,643,226 pull requests created by 142,902 Codex users. See Figure OA-2 for the distribution of event dates.

[15]On May 16, 2025, OpenAI announced the launch of a research preview of Codex, based on a finetuned version of OpenAI o3. https://en.wikipedia.org/wiki/OpenAI_Codex

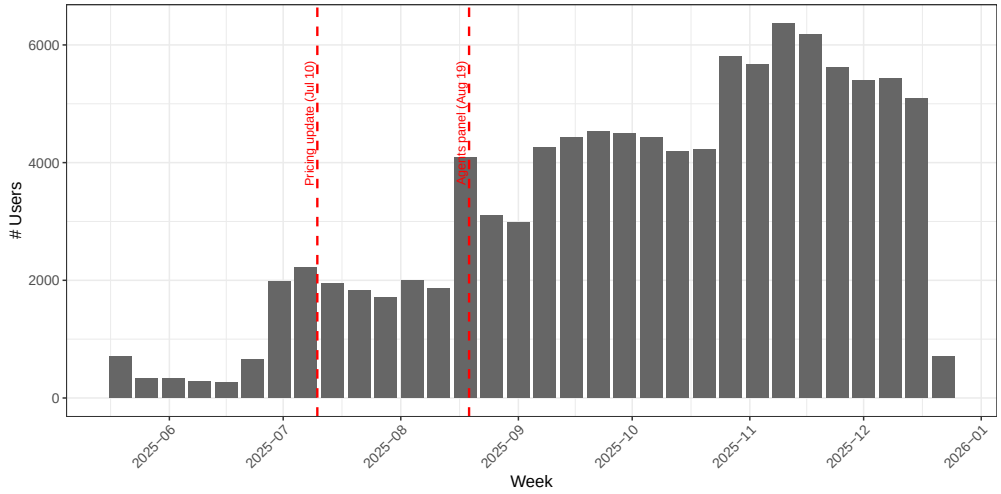


Figure OA-1: Event date of GitHub Async Agent (Copilot Agent) adoption

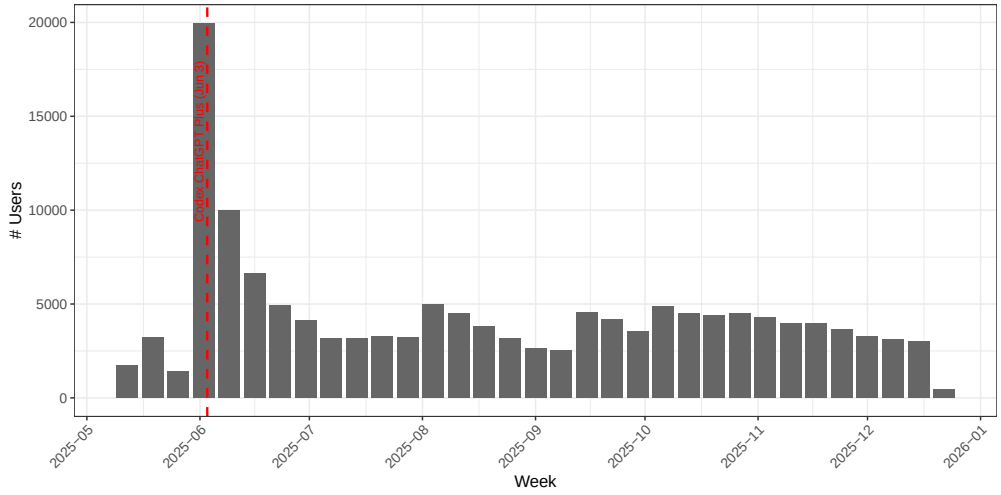


Figure OA-2: Event date of Codex Async Agent adoption

Claude Code Sync Agent When starting a conversation, Claude Code automatically pulls `CLAUDE.md` files into its context[16]. 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[17].

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

[16] https://www.anthropic.com/engineering/claude-code-best-practices

[17] 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 (https://www.gharchive.org/), 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.

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.

We identify 59,614 `CLAUDE.md` files across 51,865 repositories, along with 179,879 associated commits authored by 28,558 unique authors. See Figure OA-3 for the distribution of event dates.

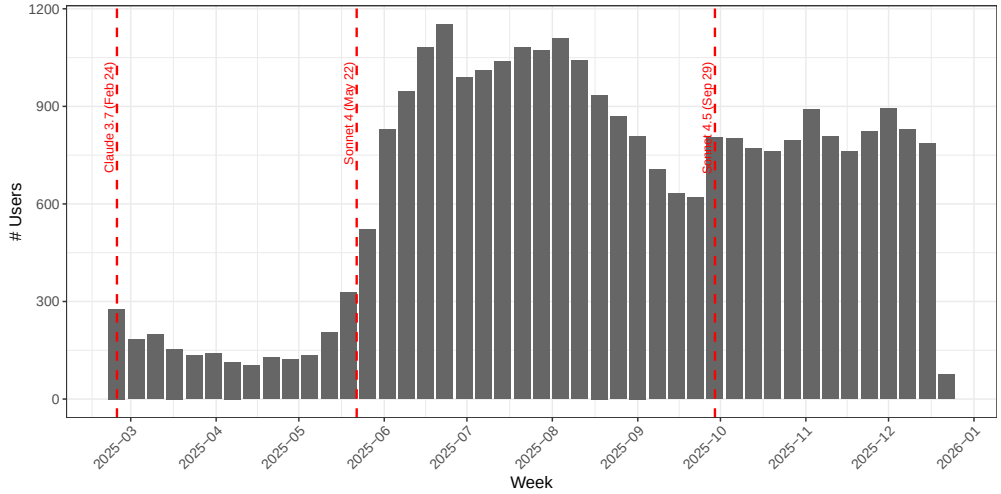


Figure OA-3: Event date of Claude Code Sync Agent adoption

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.

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.

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.

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.2.3 User-level Commit Data

Commits by Agent Users 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.

B.2.4 Panel Construction

GitHub Async Agent (Copilot Agent) Panel Construction We construct a user-week panel and decompose each user's commits into three components: human-authored commits outside Copilot pull requests, human-authored commits within Copilot pull requests, and Copilot-authored commits attributed to the user. We build the panel by integrating two data sources: overall commit activity and commits associated with Copilot agent-generated pull requests.

We drop commits with missing developer identifiers and deduplicate by commit URL. For commits in Copilot pull requests, we split the data into bot-authored commits (author ID 198982749) and human-authored commits. For bot-authored commits, which represent code written by the AI agent, we identify the corresponding human developers through co-author information embedded in commit messages. We explicitly exclude two non-human identities from the co-author list: the Copilot bot itself (ID: 198982749) and the GitHub Copilot account (ID: 175728472).

We apply two cleaning procedures to the bot-authored commits. First, we remove commits with messages "Initial plan for issue" or "Initial plan," as these represent automated planning activities rather than code contributions. Second, for bot-authored commits lacking co-author information, we implement pull-request-level imputation logic: if all other commits in the same pull request share exactly one unique co-author, we impute that co-author to the missing observation. We drop observations if other commits in the pull request list multiple distinct co-authors (ambiguous attribution) or if no other commits contain co-author information (unavailable attribution).

We assign each commit to a calendar week using a sequence of Monday dates aligned

with the previous year for seasonal matching purposes. We remove commits flagged as occurring within pull requests prior to 2025-05-19, the public preview release date of the Copilot Coding Agent. We then aggregate to the user-week level, computing four distinct commit counts for each developer in each week:

1. Commits that appear only in general commit collection and not in any Copilot pull request (human-authored commits outside Copilot pull requests)
2. Commits that appear in Copilot pull requests and were originally authored by humans (human-authored commits within Copilot pull requests)
3. Commits that appear in Copilot pull requests and were originally authored by the bot but attributed to human co-authors (Copilot-authored commits)
4. Total commits across all categories

Finally, we expand the panel to be balanced in calendar time by adding zeroes for weeks with no recorded commits. We then replace the leading and trailing zeroes for each user with `NA` to indicate inactivity.

Codex Async Agent Panel Construction Because Codex pull requests and the associated commits are attributed to the developer's GitHub account, we cannot separately distinguish human-authored commits within Codex pull requests from Codex-generated commits. Accordingly, our panel decomposes each user's total commits into commits outside Codex pull requests and commits within Codex pull requests.

We combine overall commit activity data with commits associated with Codex pull requests, label each observation as occurring inside or outside a Codex pull request, merge the two datasets, and deduplicate by commit URL. When duplicates arise, we retain the within-pull-request observation. We drop pull-request commits dated before 2025-05-16, the Codex release date. Using the same week identifier aligned with the previous year, we construct a Codex user-week panel that records each user's weekly total commits and decomposes them into commits outside Codex pull requests and commits within Codex pull requests.

Claude Code Sync Agent Panel Construction For Claude, pull requests are not defined in our setting; accordingly, we construct only a user-week panel of each user's total commits. We drop observations with missing user IDs or commit dates and deduplicate by commit URL. We compute each user's weekly commit counts using the same week alignment convention as in the other panels.

GitHub Sync Agent Panel Construction For GitHub Sync Agent, we construct a user-week panel of total commits using the same methodology as for Claude Code Sync Agent. The panel covers weeks aligned with the 2024 calendar for seasonal matching with the control period.

GitHub Autocomplete and GitHub Pro Panel Construction For GitHub Autocomplete and GitHub Pro, we construct user-week panels of total commits using subscription data from Microsoft. The panels span the subscription period (2022 for Autocomplete and Pro) and are aligned with the previous year's Monday dates for seasonal matching with control periods.

Dockerfile Panel Construction For the Dockerfile placebo, we construct a user-week panel of total commits using the same methodology as for Claude Code. The panel covers the observation period with weeks aligned to the previous year for seasonal matching.

B.3 Estimation Details

B.3.1 Overview of Matching Strategies

To address potential selection bias and control for time-varying confounds, we construct matched control groups using procedures that combine exact matching on temporal dimensions with nearest-neighbor matching on pre-treatment commit activity. We employ two distinct matching strategies depending on how treatment is identified, reflecting differences in data availability across coding agents.

Our matching approach varies based on whether treatment is identified through *direct observation of tool usage* or through *subscription records*:

Strategy A: Exact Same-Day PR Matching For coding tools where treatment is identified by observing specific file patterns in pull request content (Claude Code, Codex, GitHub Agent, and Dockerfile), we observe the precise date of first tool-assisted code submission. Controls are matched to treated developers on the *exact same calendar date* (shifted by one year), ensuring precise temporal alignment. This strategy is used for:

- **Claude Code Sync Agent:** Treatment identified from commits containing `CLAUDE.md`
- **Codex Agent:** Treatment identified from PRs created by Codex
- **GitHub Async Agent:** Treatment identified from PRs created by GitHub agent
- **Dockerfile:** Treatment identified from commits containing Dockerfile files

Strategy B: Same-Week Commit-and-PR Activity Matching For agents where treatment is identified through subscription records rather than direct usage observation (GitHub Autocomplete, Github Sync Agent and GitHub Pro), the subscription date may not coincide with actual coding activity in public repositories. To ensure that both treated and control developers are actively coding at their respective event dates, we impose additional requirements: treated developers must have both commits *and* pull requests during the event week (week 0), and controls must satisfy the same condition.

B.3.2 Control Pool Construction

The control pool comprises GitHub developers in our user universe who created pull requests during the relevant donor period but never adopted any of the coding tools under study. We assemble pull request data using the GitHub REST API. The donor period varies by agent and is shifted one year prior to the treated period.

Eligibility requirements differ between the two matching strategies.

Strategy A: Exact Same-Day PR Matching For each candidate control developer j and each date d on which they created a pull request (the potential pseudo-treatment date), we:

1. Map date d to its corresponding week start w_0 (the Monday of the week containing d) using ISO week conventions.
2. Define week -11 as the Monday 77 days prior to w_0.
3. Verify that developer j has at least one observation (possibly with zero commits) in the commit panel for week w_0, ensuring they were active on GitHub during that week.
4. Verify that developer j has at least one week with strictly positive commits at or before week -11, ensuring measurable pre-treatment commit activity for matching.

A control developer is eligible for matching on date d only if both conditions (3) and (4) hold.

Strategy B: Same-Week Commit-and-PR Activity Matching For agents with subscription-based treatment identification, we impose stricter eligibility requirements to ensure observed coding activity. For each candidate control developer j and each week w in the matching window:

1. Verify that developer j has strictly positive commits during week w.

2. Verify that developer j created at least one pull request during week w.

3. Verify that developer j has at least one week with strictly positive commits at or before week -11 (77 days prior), ensuring measurable pre-treatment activity.

A control developer is eligible for matching in week w only if all three conditions hold. This dual requirement (commits *and* PRs) ensures that matched controls were actively engaged in substantial coding work during the pseudo-treatment week, parallel to the activity requirement imposed on treated developers whose subscription dates may not reflect actual tool usage.

For treated developers under Strategy B, we additionally require that they have both commits and pull requests during their treatment week (week 0). This filter ensures that we analyze developers who were actively coding when their subscription began, rather than developers who subscribed but may not have been coding during that period.

B.3.3 Matched Pair Selection

We match treated and control developers using a sequential greedy algorithm with randomized ordering. The algorithm is identical for both strategies, differing only in the eligibility sets:

1. **Randomization:** Randomly shuffle the order of treated developers to ensure 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 control developers $\mathcal{C}_i$ from the pre-computed eligibility table:

   - *Strategy A:* Controls with PR on the exact same calendar date one year prior
   - *Strategy B:* Controls eligible during the same calendar week one year prior

   (c) Remove any control developer who has already been matched (matching without replacement).

   (d) For each eligible control $j \in \mathcal{C}_i$, compute the absolute difference in pre-treatment commits: $\Delta_{ij} = \left|\text{PreCommits}_i - \text{PreCommits}_j\right|$.

   (e) Select the control j^* with the smallest difference (nearest neighbor): $j^* = \arg\min_{j \in \mathcal{C}_i} \Delta_{ij}$.

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

For all agents, we employ *seasonal alignment* by matching across consecutive years. Treated developers in year Y are matched to controls in year $Y-1$.

This one-year shift ensures that seasonal patterns in developer activity (e.g., holiday effects, academic calendars, quarterly business cycles) are balanced between treatment and control groups. The seasonal alignment is critical given that treatment dates for some agents span multiple months, during which productivity patterns may vary systematically.

Table OA-3 summarizes the key matching parameters for each coding agent.

Table OA-3: 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." indicates what activity is required at the pseudo-treatment date/week: "PR only" means that we impose no additional restrictions on treated users beyond the event definition, and for control users, it is sufficient to have a pull request on the corresponding date; "Commit + PR" means having both commits and pull requests during that week is required for both treated and control developers.

B.3.4 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:

$$Y_{it} - Y_{it}^{control} = \sum_{k=-11, k\neq -1}^{24} \beta_k \cdot \mathbf{1}\{t - t_i^* = k\} + \gamma_{pre} \cdot \mathbf{1}\{t < t_i^* - 11\} + \gamma_{post} \cdot \mathbf{1}\{t > t_i^* + 24\} + \alpha_i + \varepsilon_{it} \tag{26}$$

where t_i^* denotes the treatment event week for developer i, $\mathbf{1}\{\cdot\}$ is the indicator function, and α_i captures developer 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:

$$\widetilde{Y}_{it} = \frac{Y_{it}}{\bar{Y}_i^{pre}}, \quad \widetilde{Y}_{it}^{control} = \frac{Y_{it}^{control}}{\bar{Y}_i^{pre,control}}, \quad \text{where} \quad \bar{Y}_i^{pre} = \frac{1}{11}\sum_{k=-11}^{-1} Y_{i,t_i^*+k} \tag{27}$$

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

The normalized event study specification is:

$$\widetilde{Y}_{it} - \widetilde{Y}_{it}^{control} = \sum_{k=-11, k\neq -1}^{24} \beta_k \cdot \mathbf{1}\{t - t_i^* = k\} + \gamma_{pre} \cdot \mathbf{1}\{t < t_i^* - 11\} + \gamma_{post} \cdot \mathbf{1}\{t > t_i^* + 24\} + \alpha_i + \varepsilon_{it} \tag{28}$$

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:

$$\tilde{\beta}_k = \hat{\beta}_k - \frac{1}{11}\sum_{j=-11}^{-1} \hat{\beta}_j \tag{29}$$

B.3.5 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:

$$Y_{it} = \sum_{k=-11}^{24} \beta_k \cdot \mathbf{1}\{t - t_i^* = k\} + \varepsilon_{it} \tag{30}$$

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 27), rather than the async outcome's own pre-period mean (which is mechanically zero), and report this normalized version.

B.3.6 *Post-Period Binned Regression*

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

$$\begin{aligned}\widetilde{Y}_{it} - \widetilde{Y}_{it}^{control} = \quad & \beta_0 \cdot \mathbf{1}\{t - t_i^* = 0\} \\ & + \beta_{1\text{-}8} \cdot \mathbf{1}\{1 \le t - t_i^* \le 8\} \\ & + \beta_{9\text{-}16} \cdot \mathbf{1}\{9 \le t - t_i^* \le 16\} \\ & + \beta_{17\text{-}24} \cdot \mathbf{1}\{17 \le t - t_i^* \le 24\} \\ & + \gamma_{pre} \cdot \mathbf{1}\{t < t_i^* - 11\} + \gamma_{post} \cdot \mathbf{1}\{t > t_i^* + 24\} + \alpha_i + \varepsilon_{it} \end{aligned} \tag{31}$$

The omitted category is the pre-period (weeks −11 to −1). The coefficient $\beta_{1\text{-}8}$ represents the average treatment effect during weeks 1–8 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 27):

$$\begin{aligned}\frac{Y_{it}}{\bar{Y}_i^{pre}} = \quad & \beta_0 \cdot \mathbf{1}\{t - t_i^* = 0\} + \beta_{1\text{-}8} \cdot \mathbf{1}\{1 \le t - t_i^* \le 8\} \\ & + \beta_{9\text{-}16} \cdot \mathbf{1}\{9 \le t - t_i^* \le 16\} + \beta_{17\text{-}24} \cdot \mathbf{1}\{17 \le t - t_i^* \le 24\} \\ & + \gamma_{pre} \cdot \mathbf{1}\{t < t_i^* - 11\} + \gamma_{post} \cdot \mathbf{1}\{t > t_i^* + 24\} + \varepsilon_{it} \end{aligned} \tag{32}$$

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

B.3.7 *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 $j \in \{\text{Claude, GitHub Agent, Codex, GitH}$ index the coding agent. The pooled matched specification is:

$$\widetilde{Y}_{ijt} - \widetilde{Y}_{ijt}^{control} = \sum_{k=-11, k\neq -1}^{24} \beta_k \cdot \mathbf{1}\{t - t_{ij}^* = k\} + \gamma_{pre} \cdot \mathbf{1}\{t < t_{ij}^* - 11\} + \gamma_{post} \cdot \mathbf{1}\{t > t_{ij}^* + 24\} + \alpha_{ij} + \varepsilon_{ijt} \tag{33}$$

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 30 and 32) are used with standard errors clustered at the developer-by-agent level but without fixed effects.

B.3.8 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:

$$\bar{M}_i^{pre} = \frac{1}{2}\left(\bar{Y}_i^{pre} + \bar{Y}_i^{pre,control}\right) \tag{34}$$

where $\bar{Y}_i^{pre}$ and $\bar{Y}_i^{pre,control}$ are the treated and control pre-period means from equation 27. 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 31. For matched outcomes, the quartile interaction specification is:

$$\begin{aligned}\widetilde{Y}_{it} - \widetilde{Y}_{it}^{control} = \quad & \sum_{q=1}^{4}\sum_{b\in\mathcal{B}} \beta_{q,b} \cdot Q_q(i) \cdot \mathbf{1}\{t - t_i^* \in b\} \\ & + \gamma_{pre} \cdot \mathbf{1}\{t < t_i^* - 11\} + \gamma_{post} \cdot \mathbf{1}\{t > t_i^* + 24\} + \alpha_i + \varepsilon_{it}\end{aligned} \tag{35}$$

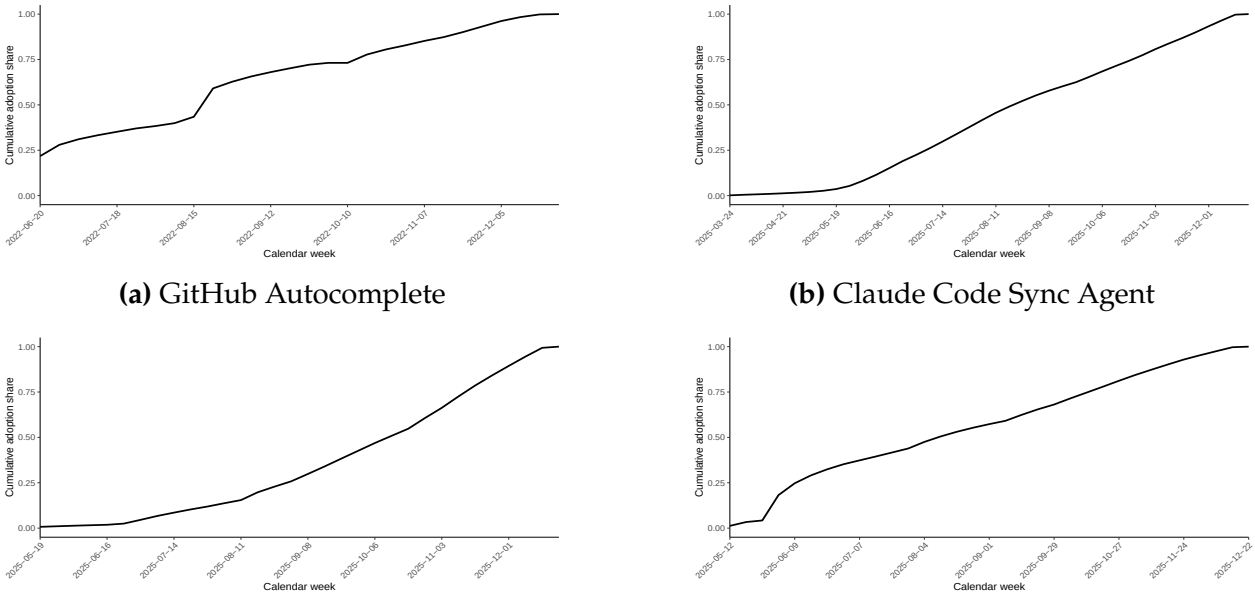
where $Q_q(i) = 1$ if developer i belongs to quartile q, and $\mathcal{B} = \big\{\{0\},\ \{1,\ldots,8\},\ \{9,\ldots,16\},\ \{17,\ldots,24\}\big\}$ indexes the post-period bins. The omitted category is the full pre-period (weeks -11 to -1) for all quartiles. Standard errors are clustered at the developer 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 $100 \times \beta_{q,\text{17-24}}$ as the percentage treatment effect for quartile q in weeks 17–24. 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.

B.3.9 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

Figure OA-4: Adoption of AI Coding Tools



(a) GitHub Autocomplete

(b) Claude Code Sync Agent

(c) GitHub Async Agent

(d) Codex Async Agent

Notes: This figure shows the distribution of adoption events over time for each AI coding tool. Each panel plots the number of new adopters by calendar date. We can compute but omit the distribution of GitHub Sync Agent events as it is considered sensitive information.

post-period positive values. All regressions are estimated using the `fixest` package in R.

C Descriptive Facts about Usage of AI Coding Tools

In this section, we present descriptive facts on the usage of AI coding tools. We begin by documenting adoption patterns for each tool, focusing on the cumulative adoption rate in our sample since the tool's release. We conclude the section by analyzing the stickiness of usage, capturing how persistently developers continue to use these tools after adoption.

C.1 Adoption

Figure OA-4 shows the cumulative adoption of each coding tool over time among developers in our analysis sample. We do not show adoption for the GitHub Sync Agent for confidentiality reasons. Importantly, these adoption patterns do not represent population-level product adoption, but rather the timing of adoption within our sample. Nevertheless, they are informative for understanding our event studies when measuring productivity effects and provide useful descriptive evidence on the diffusion of AI coding tools, though this evidence should be interpreted with caution.

As shown in Figure OA-4(a), adoption of GitHub Autocomplete exhibits an immediate

jump following release and continues to increase rapidly over time. This pattern reflects the relatively low barrier to adoption: autocomplete integrates seamlessly into existing workflows and requires minimal behavioral adjustment. Developers can begin using autocomplete without fundamentally changing how they write code, which facilitates rapid diffusion.

In contrast to autocomplete, adoption of both sync and async agents follows a less rapid initial trajectory. As shown in Figure OA-4(b) and Figure OA-4(c), adoption of the Claude Code synch agent and the GitHub asynch agent is initially modest—particularly in the first several weeks following release—but increases gradually over time, producing diffusion patterns that more closely resemble the S-shaped adoption curves commonly documented in the diffusion literature.

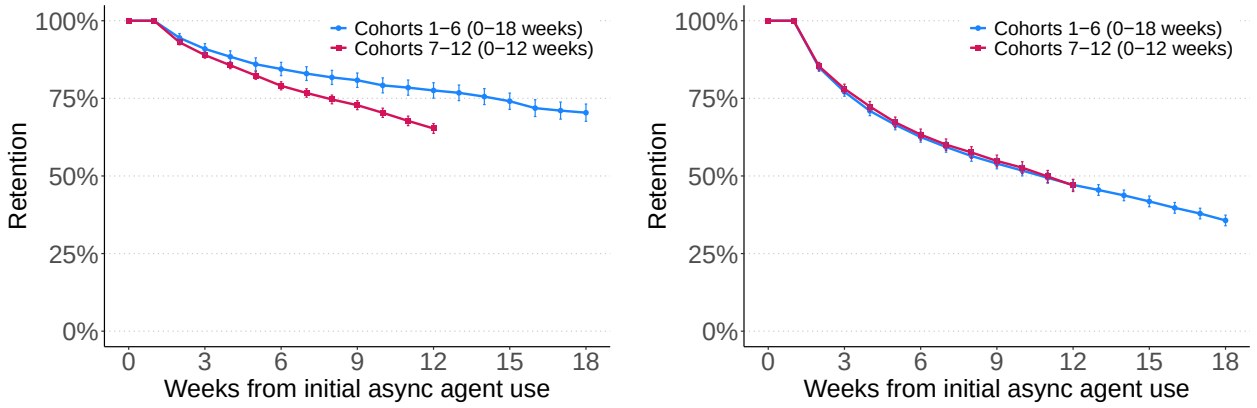
The adoption pattern for the Codex async agent, shown in Figure OA-4(d), differs in an important respect. Codex exhibits a sharp initial increase in adoption, followed by a more linear growth phase. One possible explanation is the initial attention surrounding the release of OpenAI's first dedicated coding product, which may have prompted many developers to experiment with the tool early on. By contrast, GitHub and Claude Code were introduced into environments where developers were already accustomed to AI-assisted coding: GitHub users had prior experience with AI coding tools, and Claude had already been widely used through third-party products such as Cursor.

C.2 Stickiness of Usage of Coding Tools

In this final subsection, we characterize post-adoption engagement with async coding agents, focusing on user retention—specifically, the extent to which developers continue using the tool after initial adoption. Retention provides a natural revealed-preference measure of perceived net benefits, as developers are unlikely to continue using a tool they find unproductive.

We measure retention using a "last-use" definition. Specifically, once a developer first uses an agent, we treat them as retained in all subsequent weeks up to (and including) their final observed agent use; retention therefore does not require weekly usage. Figure OA-5 plots the resulting retention curves against weeks since initial use. To focus on more than one-off experimentation, Figure OA-5 restricts attention to developers who use the agent in two consecutive weeks; Appendix Figure OA-9 shows the same curves without this restriction.

Two patterns stand out. First, retention is high for both products: six weeks after adoption, about 75% of Copilot users and 55% of Codex users are still retained. Second, Copilot exhibits modestly stronger retention than Codex, which may reflect tighter product



(a) Copilot Retention

(b) Codex Async Agent Retention

Figure OA-5: Retention Curve for Async Agents (Multi-Week Adoption)

Note: This figure provides user-level retention curves for GitHub Copilot's and Codex Async Agent's offerings. We define a user as retained for all periods until their last period of usage. We restrict attention to users who used the agent in two subsequent weeks. Both panels break out users by adoption cohort, revealing considerable heterogeneity for GitHub Copilot: early adopters are significantly more likely to stick with the agent than later adopters.

integration.

Finally, we report retention separately by adoption cohort. This is important because later cohorts are mechanically right-censored by the end of our sample, so their longer-run retention is not yet observed. The cohort split also reveals substantial heterogeneity for Copilot: early adopters are markedly more likely to remain retained, consistent with stronger underlying engagement with LLM-based tools among the earliest users. This gap widens further when we expand the sample to include one-week adopters (i.e., in Appendix Figure OA-9).

D Learning

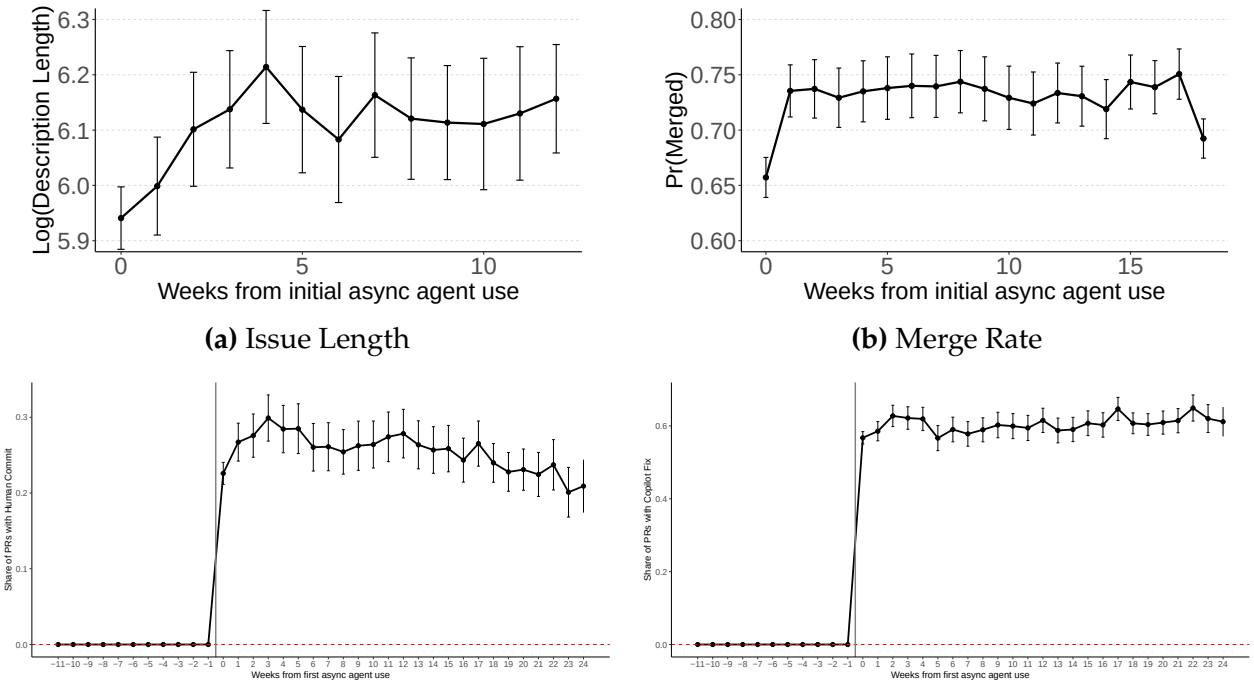
As prior work emphasizes, generative AI has a *jagged frontier*: even sophisticated users often cannot predict whether a given task is well-suited for an AI agent or beyond current model capabilities (Dell'Acqua et al., 2023). This suggests that adoption of GitHub Async Agent may involve meaningful learning. In an initial trial-and-error phase, developers may discover which parts of a project can be safely delegated to the agent and which require closer human control.

We investigate this hypothesis using two outcomes, both shown in Figure OA-6. To avoid compositional changes over event time, we compute both outcomes on a fixed sample of users observed for at least twelve weeks after their initial agent use.

First, Figure OA-6(a) plots the length of task descriptions assigned to GitHub Async Agent against weeks since adoption. Task descriptions become longer over the first month: in week 0 they average 383 characters, rising steadily to 493 characters by week 4, a 28% increase, after which they stabilize.

Second, Figure OA-6(b) reports the fraction of agent-written pull requests that are eventually merged. The merge rate rises sharply from approximately 65% in the adoption week to roughly 74% in subsequent weeks. Relative to the gradual increase in task-description length, the merge-rate pattern is more discrete, consistent with developers quickly learning to avoid mismatched tasks (or to better scope tasks) and then refining task specification more incrementally over time. Panel OA-6(c) shows the fraction of agent PRs containing a human commit, which exhibits a slight decline over time, hinting at the possibility that agents' code requires less human correction as developers learn to assign more appropriate tasks. Panel OA-6(d) examines the likelihood of multiturn conversations and shows no strong over-time pattern.

E Additional Figures



(a) Issue Length

(b) Merge Rate

(c) Human Commits

(d) Multiturn Conversations

Figure OA-6: Learning by GitHub Async Agent Users

Note: This figure exhibits the mean log number of characters in issues assigned to GitHub Async Agent by event time (Panel OA-6(a)) and the fraction of agent PRs that are merged (Panel OA-6(b)). Both panels restrict attention to users who adopt the coding agent for at least twelve weeks, ensuring no compositional effects. We find that users quickly start to write longer issues, with issues increasing 28% in length by week 4 after adoption. This may be a result of users' realization that the coding agent lacks context on the project that they are taking for granted. Panel OA-6(b) shows that the merge rate increases discretely from approximately 65% in the adoption week to roughly 74% in subsequent weeks, after which it stabilizes. Panels OA-6(c) and OA-6(d) show no strong over-time patterns in the fraction of PRs that contain a human commit (which would be indicative of a human fixing Copilot code) nor the likelihood of multiturn conversations.

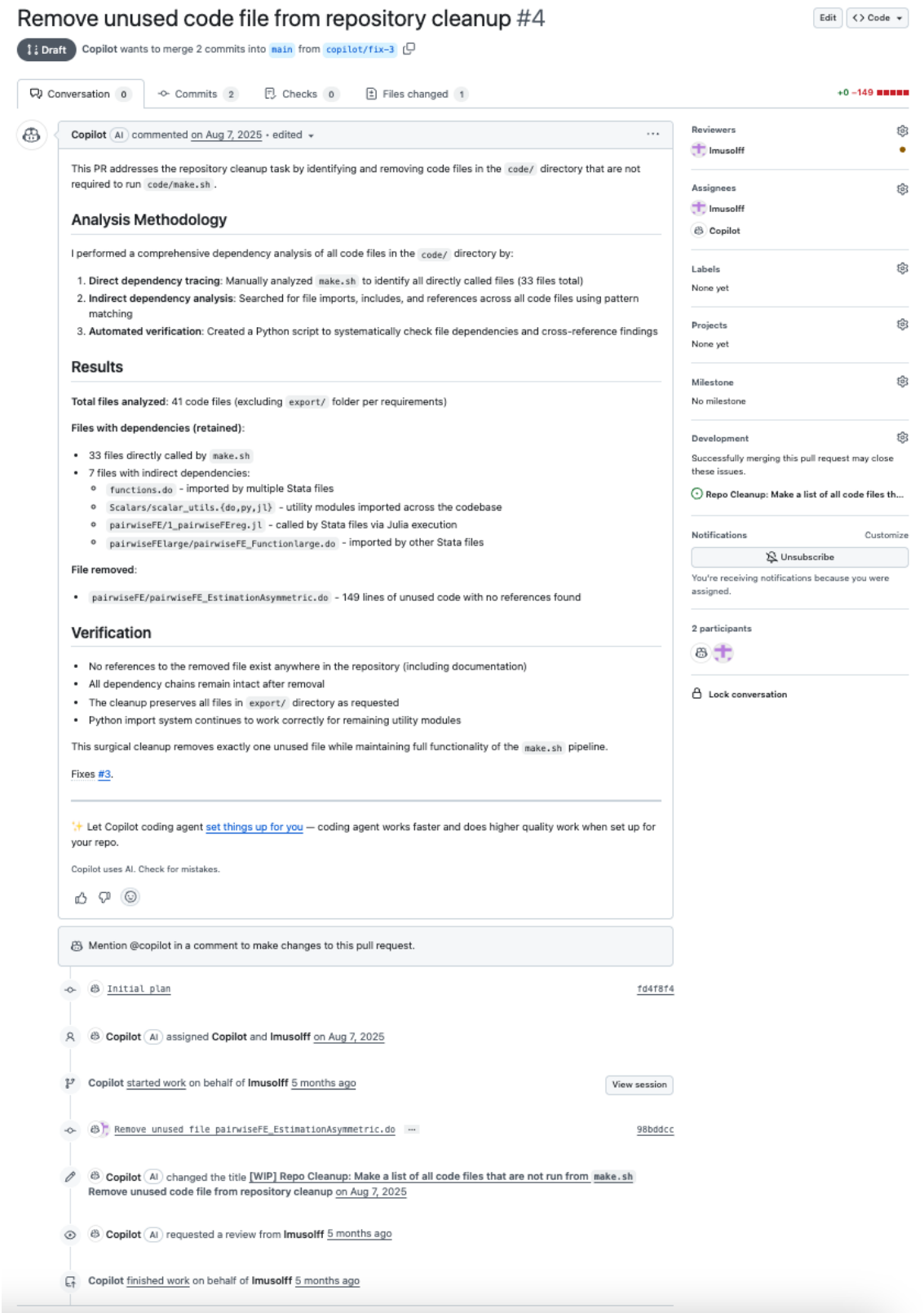


Figure OA-7: Example of a PR drafted by Copilot

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

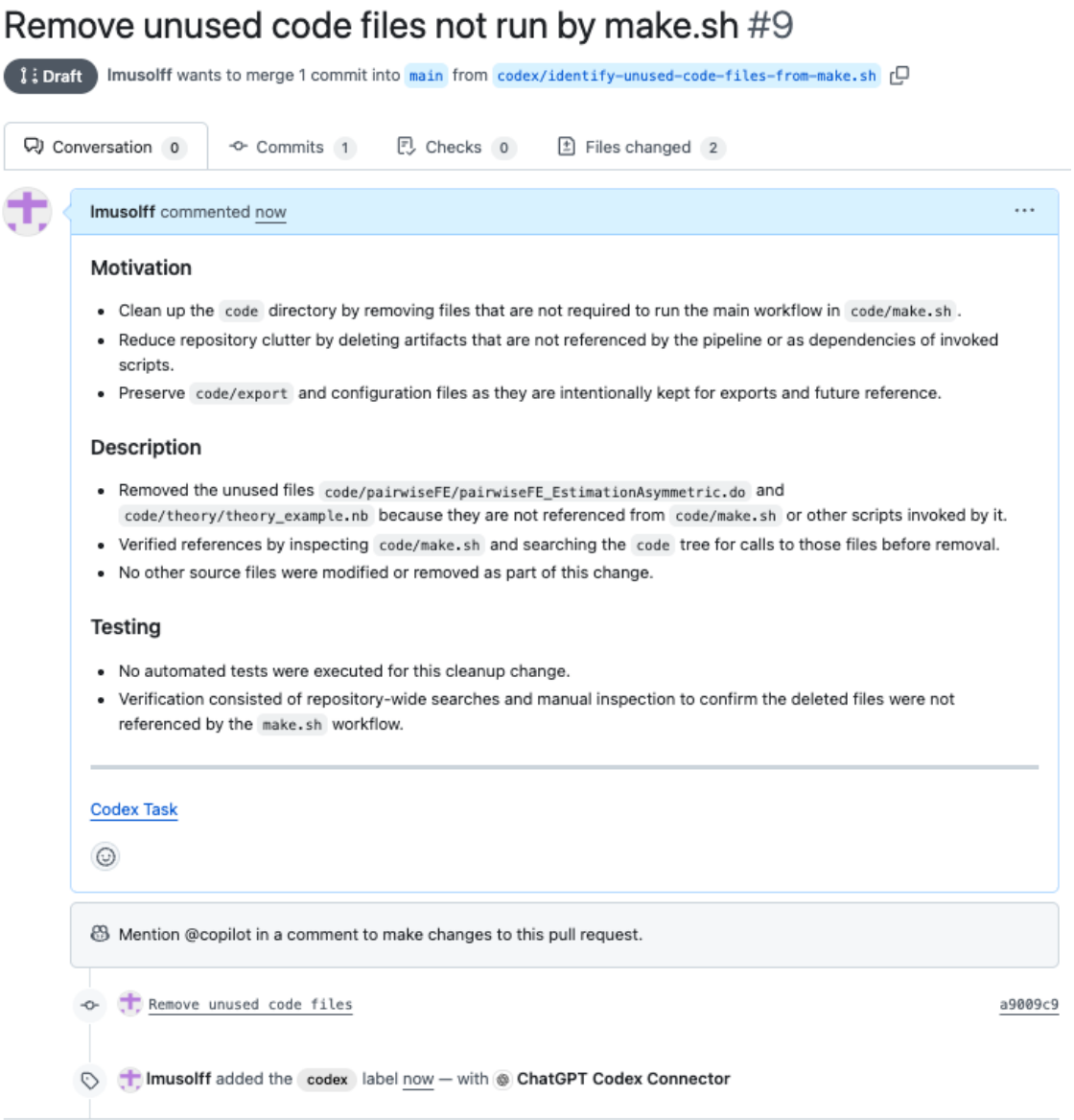
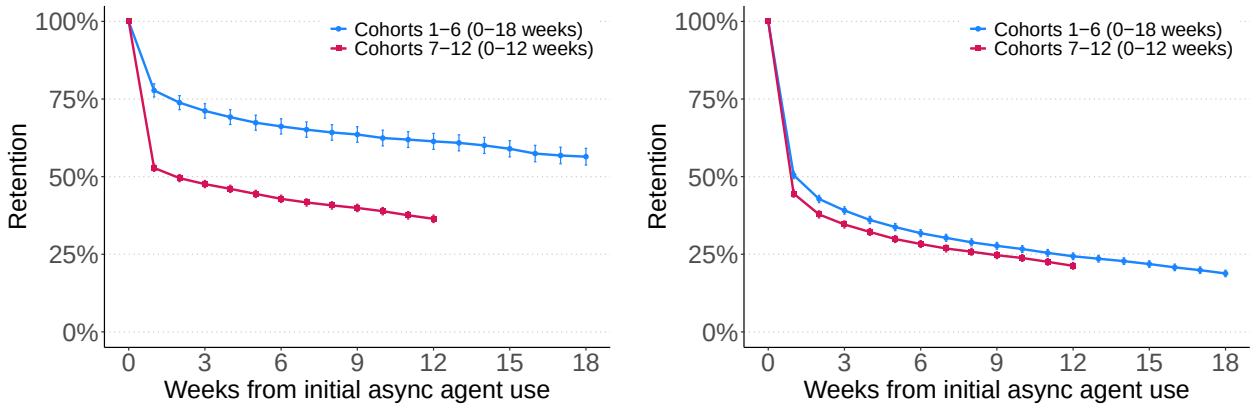


Figure OA-8: Example of a PR drafted by Codex Async Agent

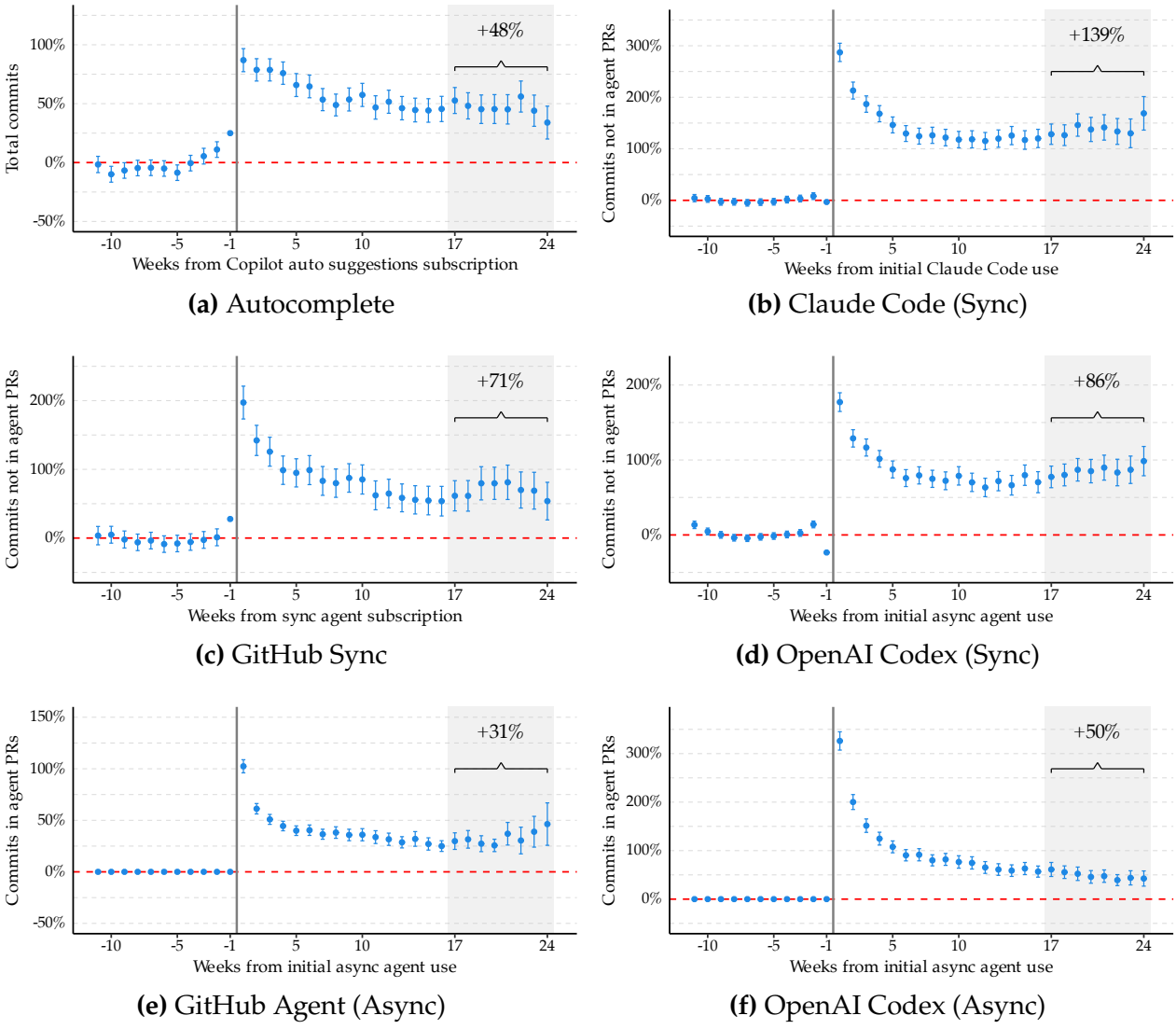
Note: This figure provides an example of a pull request that has been drafted by Codex Async Agent.



(a) Copilot Retention **(b)** Codex Async Agent Retention

Figure OA-9: Retention Curve for Async Agents

Note: This figure provides user-level retention curves for GitHub Copilot's and Codex Async Agent's offerings. We define a user as retained for all periods until their last period of usage. Both panels break out users by adoption cohort, revealing considerable heterogeneity for GitHub Copilot: early adopters are significantly more likely to stick with the agent than later adopters.



(a) Autocomplete

(b) Claude Code (Sync)

(c) GitHub Sync

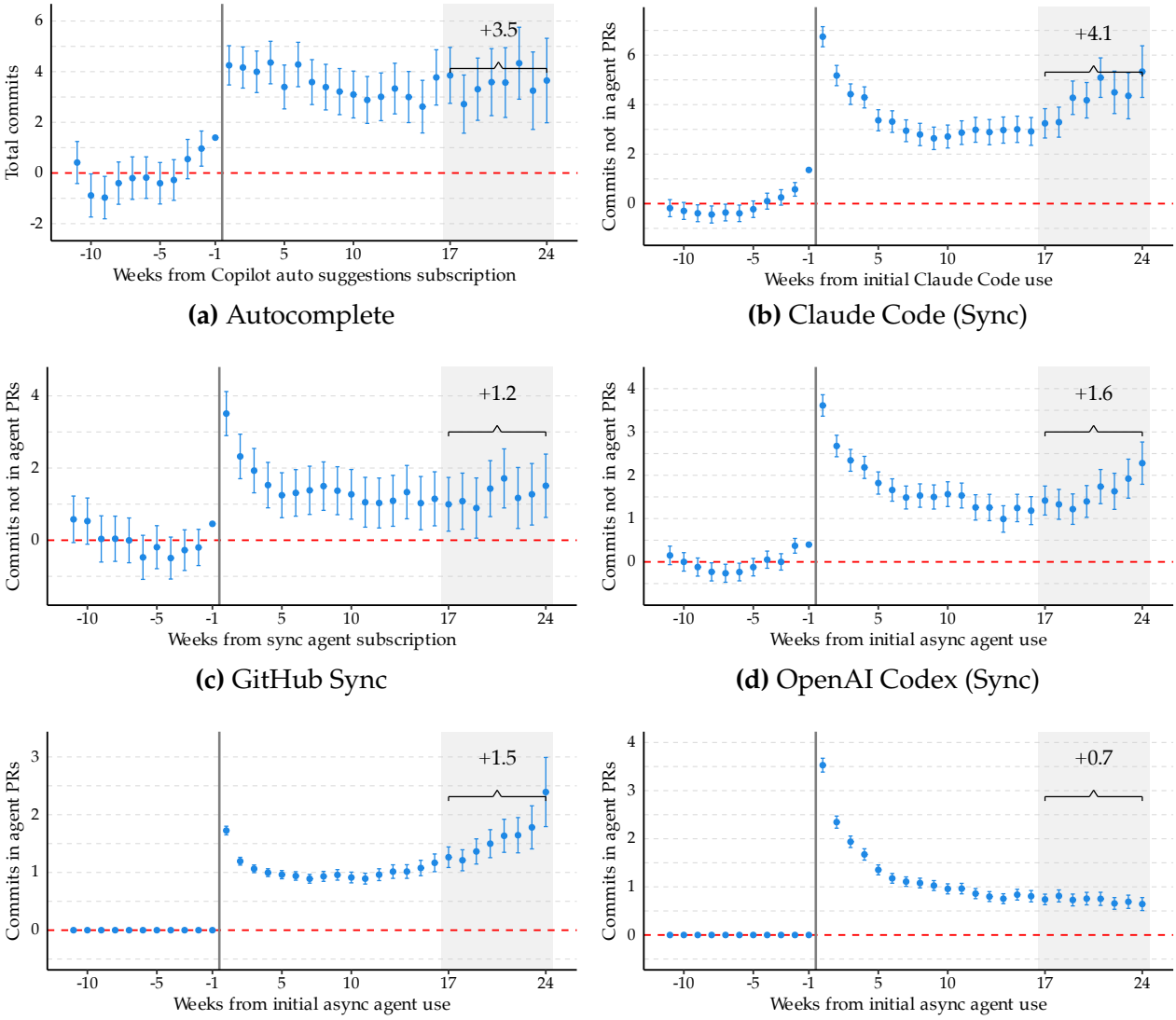
(d) OpenAI Codex (Sync)

(e) GitHub Agent (Async)

(f) OpenAI Codex (Async)

Figure OA-10: Productivity Effects by Tool (Normalized)

Note: 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.



(a) Autocomplete

(b) Claude Code (Sync)

(c) GitHub Sync

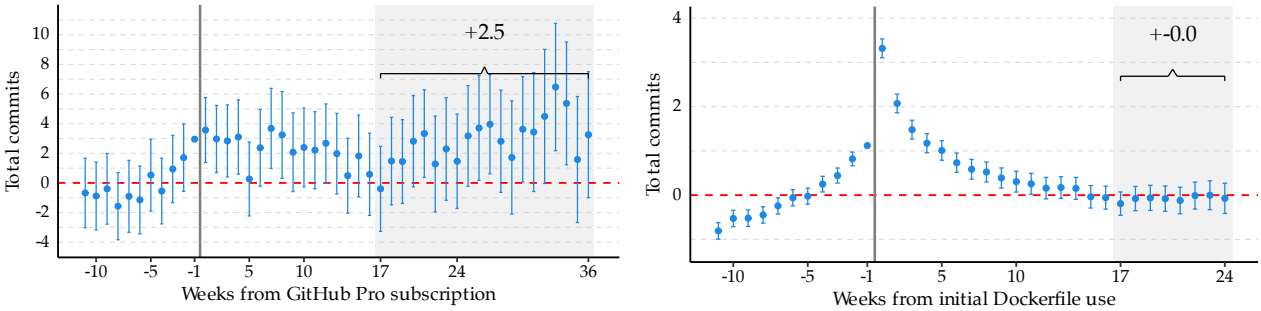
(d) OpenAI Codex (Sync)

(e) GitHub Agent (Async)

(f) OpenAI Codex (Async)

Figure OA-11: Productivity Effects by Tool (Levels)

Note: This figure presents the same per-tool event studies as Figure OA-10, but with the outcome measured in levels (raw commit counts) rather than normalized by the pre-period mean.



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

Figure OA-12: Productivity Effects of Placebo Tools (Levels)

Note: 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.

References for Online Appendix

Chirinko, R. S. (2008). σ: The long and short of it. *Journal of Macroeconomics 30*(2), 671–686. The CES Production Function in the Theory and Empirics of Economic Growth.

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 (2023, September). Navigating the jagged technological frontier: Field experimental evidence of the effects of AI on knowledge worker productivity and quality. Working Paper 24-013, Harvard Business School. Harvard Business School Working Paper No. 24-013. Available at SSRN.

Jones, C. I. (2026, January). A.I. and our economic future. Working Paper 34779, National Bureau of Economic Research.

Jones, C. I. and C. Tonetti (2026, January). Past automation and future A.I.: How weak links tame the growth explosion. Unpublished manuscript.

Raval, D. R. (2019). The micro elasticity of substitution and non-neutral technology. *The RAND Journal of Economics 50*(1), 147–167.