Measuring the Impact of Early-2025 AI on Experienced Open-Source Developer Productivity

Joel Becker*, **Nate Rush***, **Beth Barnes, David Rein**

Model Evaluation & Threat Research (METR)

Abstract

Despite widespread adoption, the impact of AI tools on software development in the wild remains understudied. We conduct a randomized controlled trial (RCT) to understand how AI tools at the February–June 2025 frontier affect the productivity of experienced open-source developers. 16 developers with moderate AI experience complete 246 tasks in mature projects on which they have an average of 5 years of prior experience. Each task is randomly assigned to allow or disallow usage of early-2025 AI tools. When AI tools are allowed, developers primarily use Cursor Pro, a popular code editor, and Claude 3.5/3.7 Sonnet. Before starting tasks, developers forecast that allowing AI will reduce completion time by 24%. After completing the study, developers estimate that allowing AI reduced completion time by 20%. Surprisingly, we find that allowing AI actually *increases* completion time by 19%—AI tooling slowed developers down. This slowdown also contradicts predictions from experts in economics (39% shorter) and ML (38% shorter). To understand this result, we collect and evaluate evidence for 20 properties of our setting that *a priori* could contribute to the observed slowdown effect—for example, the size and quality standards of projects, or prior developer experience with AI tooling. Although the influence of experimental artifacts cannot be entirely ruled out, the robustness of the slowdown effect across our analyses suggests it is unlikely to primarily be a function of our experimental design.

1 Introduction

Software development is an important part of the modern economy, and a key domain for understanding and forecasting AI capabilities [1; 2]. Frontier AI systems demonstrate impressive capabilities on a wide range of software benchmarks [3; 4; 5; 6; 7; 8; 9] and in experiments measuring AI's impact on developer productivity when completing synthetic tasks [10; 11]. However, tasks used in these *lab experiments* sacrifice realism for scale and efficiency: the tasks are typically self-contained, do not require much prior context/familiarity to understand and complete, and use algorithmic evaluation metrics which do not capture many important capabilities [12; 13; 14]. As a result, it can be difficult to draw inferences from results on these evaluations about AI's impact in practice.

To reduce the inferential gap between measurements of AI capabilities and real-world impact, one can measure the impact of AI systems in real-world settings (i.e. *field experiments*). Existing field experiments aimed at measuring AI's impact on software development measure outcomes like number of added lines of code or number of tasks completed [15; 16; 17]. However, AI systems can affect these outcomes without productivity actually increasing—for example, code can be more verbose but functionally equivalent, and tasks can be broken up into multiple smaller tasks without the total amount of work changing—making it challenging to interpret these results.

*Equal contribution. Correspondence to {nate, joel}@metr.org

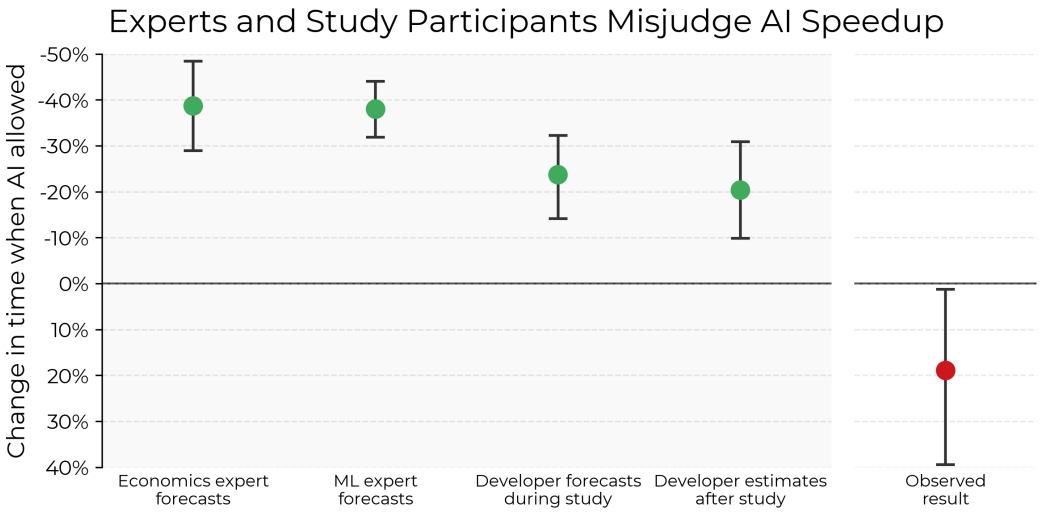


Figure 1: *Experts and study participants (experienced open-source contributors) substantially overestimate how much AI assistance will speed up developers—tasks take 19% more time when study participants can use AI tools like Cursor Pro. See Appendix D for detail on speedup percentage and confidence interval methodology.*

To directly measure the impact of AI tools on developer productivity, we conduct a randomized controlled trial by having 16 developers complete 246 tasks (2.0 hours on average) on well-known open-source repositories (23,000 stars on average) they regularly contribute to. Each task is randomly assigned to allow or disallow AI usage, and we measure how long it takes developers to complete tasks in each condition[1]. Developers, who typically have tens to hundreds of hours of prior experience using LLMs[2], use AI tools considered state-of-the-art during February–June 2025 (primarily Cursor Pro with Claude 3.5/3.7 Sonnet). We collect screen recordings as they work, providing a rich data source for analysis.

Before tasks are randomized, developers forecast that allowing AI will reduce completion time by 24%. After study participation, developers estimate that allowing AI reduced completion time by 20%. Surprisingly, we find that allowing AI actually *increases* completion time by 19%— developers are slower when using AI tooling. Figure 1 displays this observed slowdown in contrast with forecasts and post-hoc developer estimates of speedup from AI. We also collect forecasts of speedup from machine learning and economics experts in academia and industry, and find that they also substantially overestimate our observed speedup.

To understand this surprising result, we manually label 143 hours of recordings of developers' computer screens while they work (representing 29% of the total hours spent by developers), which allows us to decompose how they spend their time when working with and without AI assistance at a resolution of ~ 10 seconds. We additionally collect rich statistics from source-code management systems, interview and survey participating developers, and conduct subset analyses to better understand the nature of the slowdown result.

Using these various sources of data, we identify 20 properties of our setting and experimental design that we hypothesize *a priori* may contribute to the slowdown effect. We group these factors into four categories: a) direct productivity loss, b) experimental artifact, c) factors raising human performance, and d) factors limiting AI performance. We find evidence that 5 factors contribute to the slowdown effect, we find mixed/unclear/no evidence for 9 factors, and we find evidence against 6 factors contributing to the slowdown effect. Section 3.3 presents these factors at a high level, and Appendix C discusses each factor in detail. While we can't completely rule out the impact of exper-

[1]Crucially, the tasks are defined *before* they are randomized, limiting the impact of effects from AI assistance unrelated to productivity (e.g., more verbose but functionally equivalent code)

[2]While 93% of developers have previously used LLMs, only 44% have prior experience using the Cursor IDE.

imental artifacts, the slowdown effect appears broadly robust across a wide range of experimental design decisions.

That said, many of the factors we find evidence for contributing to slowdown are specific to the setting we study—these results do *not* imply that current AI systems are not useful in many realistic, economically relevant settings. Furthermore, these results do not imply that future models will not speed up developers in this exact setting—this is a salient possibility given the rapid pace of progress in AI capabilities recently [2]. Finally, it remains possible that further improvements to current AI systems (e.g. better prompting/agent scaffolding, or domain-specific finetuning) could yield positive speedup in this setting.

Nonetheless, our results reveal a large disconnect between perceived and actual AI impact on developer productivity. Despite widespread adoption of AI tools and confident predictions of positive speedup from both experts and developers, we observe that AI actually slows down experienced developers in this setting.

1.1 Background

Speedup, but on synthetic tasks Literature on productivity improvements on software tasks due to AI usage broadly finds that AI tools increase productivity. Peng et al. [10] and Paradis et al. [11] find 56% and 21% speedups on coding tasks when using AI assistance, and Weber et al. [18] finds a 65% increase in the rate of task requirements satisfied with AI tools. However, these studies use artificial/synthetic tasks that make it difficult to directly draw inferences about the real-world impact of AI tools. For example, Peng et al. [10] asks developers to implement a very basic HTTP server in JavaScript to satisfy several automatic test cases that are shown to the developers—this task is a) unrepresentative of most software development work, and b) likely to be similar to a large amount of LLM training data, which may unfairly advantage AI systems relative to humans.

Speedup, but with non-fixed outcome measures Other literature uses tasks found "in the wild," either via natural experiments [16] or randomized controlled trials [15; 17], finding 14-51% increases in output productivity metrics. However, these studies use outcome measures that are not fixed in advance—i.e. lines of code written, number of code commits, and pull requests[3] (PRs) as their key outcome measures respectively. It's possible for AI assistance to affect the outcomes without actually increasing productivity, e.g. by causing developers to write more verbose but functionally equivalent code, or causing them to break up pull requests into smaller chunks of work.

Because the unit of analysis is not defined before t

Impressive AI benchmark results This general consensus around AI tooling's effect on software developer productivity is perhaps unsurprising, given the impressive apparent capabilities of frontier AIs on challenging question-answering and agentic tasks used in popular AI benchmarks [19; 20].

Heterogeneous effects by experience One important question that emerges given these impressive results is whether productivity gains are captured by individuals of all experience levels. The canonical framework of Agrawal et al. [21] treats AI as a fall in the cost of prediction, with distributional consequences depending on which complementary sub-problems the tool does not solve. Existing empirical work on the micro-level effects of generative AI tools tends to find that access to these tools benefits less experienced workers more, compressing performance distributions [22; 23; 10; 24].

These heterogeneous effects motivate our focus on highly skilled open-source developers, as there has been relatively less research in this setting.

Mixed speedup results in other domains Some literature measures the impact of frontier AI systems in settings other than software development, for example, for CBRN uplift risk assessment, finding mixed results with recent AI systems [25; 26; 27; 28]. Other research finds substantial productivity increases in non-software domains [22; 23].

[3] See Appendix F for a primer on open-source development terminology.

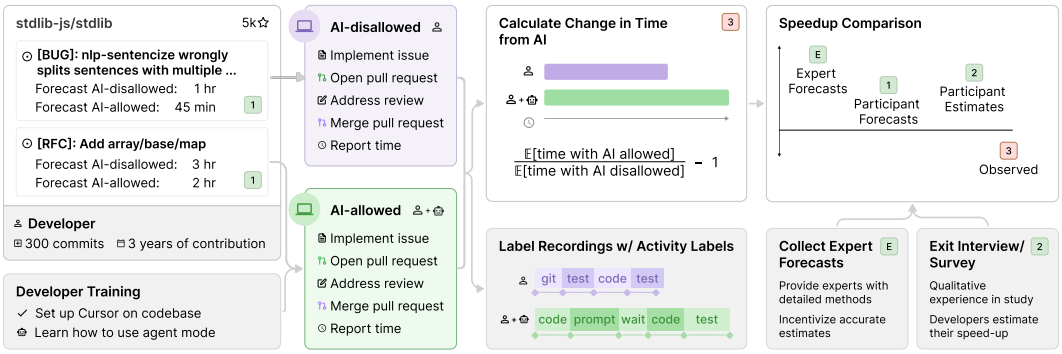


Figure 2: *Our experimental design. Tasks (referred to as issues) are defined before treatment assignment, screen recordings let us verify compliance (and provide a rich source of data for analysis), and forecasts from experts and developers help us measure the gap between expectations and observed results.*

Understanding AI's economic impact Finally, some literature tries to predict how AI capability advances might a) affect the rate of AI progress (e.g. if AI systems can substitute for human AI R&D labor), or b) broadly impact the economy. Leibowich et al. [29] interview AI researchers about how full automation of AI R&D might alter the pace of advancement, several papers explore the possibility of explosive economic growth via large-scale AI labor substitution [30; 31; 32], and the economics literature includes both optimistic and skeptical perspectives on AI's productivity impact [33; 34; 35].

Our study primarily complements existing literature measuring the impact of AI on software development by:

1. Testing AI models at the February–June 2025 frontier,
2. Using unfiltered, "live" open-source repository tasks rather than synthetic or cherry-picked tasks,
3. Using a fixed outcome measure (speedup on tasks defined before randomized treatment assignment),
4. Recruiting experienced engineers with years of expertise in the target repositories, and
5. Collecting rich data on time usage, AI code suggestions, and developers' qualitative experiences.

2 Methodology

2.1 Developers and Repositories

We recruit experienced developers from large open source repositories to work on real tasks defined on these repositories. Developers come from a mix of our professional networks and from outreach to active contributors to large, popular Github repositories. The developers are experienced software engineers (typically over a decade of experience), and are regular contributors to the repositories we use—on average, they have 5 years of experience working on their repository, representing 59% of that repository's lifetime, over which time they have made 1,500 commits to the repo. As an incentive to participate, we pay developers $150/hour. Appendix G provides more detail about our recruitment and incentivization process.

The repositories themselves are large and mature. On average, they have 23,000 stars, 1,100,000 lines of code, 4,900 forks, 20,000 commits, and 710 committers, and they broadly have very high quality bars for code contributions. For example, one set of repository contribution guidelines concludes: *"Phew. While the above may be a lot to remember [..] the motivation for enforcing process is to ensure that all code contributions meet a certain quality threshold."* Section G.7 details further statistics about individual developers and repositories.

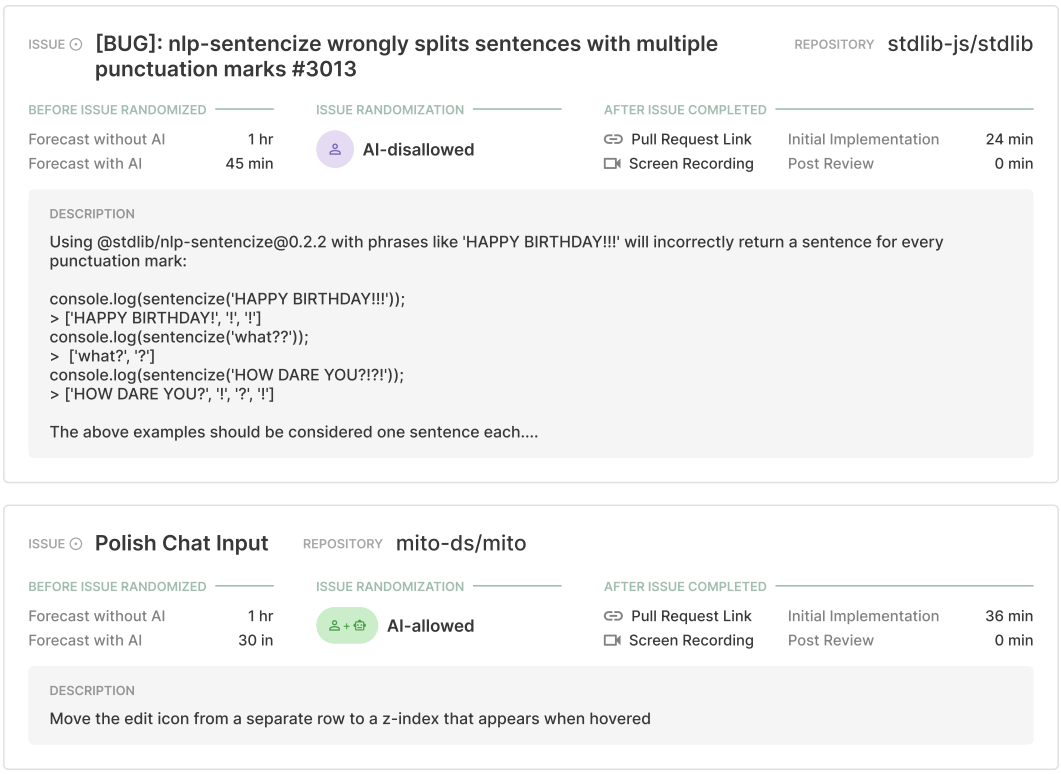


Figure 3: *Real issues completed during the study from the* `stdlibjs` *and* `mito` *repositories*

2.2 Experimental Design

Each developer provides a list of real issues in their repository to work on as part of this study. Issues are typically bug reports, feature requests, or work items used to coordinate development. They range from brief problem descriptions to detailed analyses and represent work ranging from minutes to hours. Two example issues are shown in Figure 3. Many issues are defined before the study period begins, but some are created during the study period.[4]

After collecting this issue list, developers forecast how long each issue would take if they were to complete it both with and without AI assistance. We use these forecasts as a proxy for issue difficulty, and to measure per-issue speedup anticipated by the developer. These issues are then randomized to one or the other condition via a simulated fair coin flip.[5] If AI is allowed, developers can use any AI tools or models they choose, *including no AI tooling if they expect it to not be helpful*. If AI is not allowed, no generative AI tooling can be used.[6]

Developers then work on their assigned issues in their preferred order—they are allowed to flexibly complete their work as they normally would, and sometimes work on multiple issues at a time. After completing an issue to their satisfaction, they submit a pull request (PR) to their repository, which is typically reviewed by another developer. They make any changes suggested by the PR reviewer,

[4] About half of issues included in the study were not formally defined on the repository, and instead were tracked separately for our experiment. Importantly, all issues represent real work that developers wanted to contribute to their repositories, and all work completed by developers is submitted and reviewed through each repository's standard source-code management system (e.g. GitHub/GitLab). Developers are asked to contribute issues taking a maximum of two hours, or to break up issues taking longer into multiple issues.

[5] 25 issues early in the study were randomized differently. We show that results are not sensitive to inclusion/exclusion of these issues in Section C.3.4, and we include these issues for statistical power.

[6] AI-based tab autocomplete is disallowed in the AI-disallowed condition if it uses LLMs (e.g. GitHub Copilot) but allowed otherwise. Search engines, which sometimes use AI under the hood, remain allowed in the AI-disallowed condition.

and merge their completed PR into the repository[7]. As the repositories included in the study have very high quality and review standards, merged PRs rarely contain mistakes or flaws. Finally, they self-report how long they spend working on each issue before and after PR review.

See Section G.2 for the full written instructions given to developers before they start working.

2.2.1 AI Tools and Training

Two popular means of using modern large language model (LLM) based AI tools are via web-based user interfaces (e.g. chatgpt.com) and the integrated development environment (IDE) Cursor Pro (which we provide a subscription for). Cursor is a fork of the widely used VSCode IDE with near-identical features, that additionally includes extra AI features like a language model chat interface, and an AI agent tool that can search and edit files, run arbitrary bash commands, prompt/ask the user for more details when relevant, and iterate/debug programs without constant input from users. Developers have a range of experience using AI tools: 93% have prior experience with tools like ChatGPT, but only 44% have experience using Cursor.

We provide developers with Cursor Pro subscriptions and conduct live basic training, validating that developers are able to prompt Cursor effectively to edit files in their own codebase, accept changes, and revert to previous checkpoints. However, we don't require that they use Cursor specifically. Developers working on issues for which AI is allowed can use any AI tools of their choosing, or no AI tools if they prefer. See Section F.2 for further information on these two methods of accessing AI assistance, and Appendix G for more detail about our training and onboarding process.

2.2.2 Data Collection

Contributors completed issues largely as they would outside of our experiment, with a few exceptions: they typically record their screen as they work (providing us with a source of rich data on their AI usage and working patterns), when using AI they often use the Cursor IDE, which sometimes differs from their normal development environment (e.g. neovim, PyCharm), and they record details of their working patterns after each issue is completed. Broadly, developers reported that they were not significantly inconvenienced or affected by these differences compared to their normal workflows (see section Section 3.3 for discussion of how experimental design may affect results). Developers self-report the length of time they spend working on each issue before and after PR review.

2.3 Effect Estimation

We aim to measure the percentage change in issue time to completion due to AI, defined as:

$$S = \frac{\mathbb{E}[\text{completion time with AI allowed}]}{\mathbb{E}[\text{completion time with AI disallowed}]} - 1$$

A value of -50% would indicate issues assigned to AI-allowed taking half the time of issues assigned to AI-disallowed; 0% would indicate that issues take the same time to complete with and without AI-allowed; 100% would indicate that issues assigned to AI-allowed take twice as long as issues assigned to AI-disallowed. For convenience and clarity, we typically report the absolute value of S, and colloquially refer to negative values of as *speedup*, and positive values as *slowdown*[8]

We use the total implementation time (including both pre- and post-pull request review time) developers self-report spending on each issue to estimate the percentage change in issue completion time due to AI. A small number of issues do not end up getting reviewed/merged for various reasons—we impute the post-review implementation time of these issues with the mean post-review time per treatment condition. Section C.3.4 shows that our results are robust to the specific imputation method used.

We estimate S via a log-linear regression for total implementation time, incorporating estimates of issue difficulty to reduce noise in our estimate. We also use adapted versions of this regression when

[7]Not all PRs end up getting merged by the end of the study period—see Section C.3.4 for discussion of how this may affect results.

[8]This is a mild abuse of language, because speedup and slowdown refer to *rates*, while the quantity we actually estimate and report is the percentage change in time from AI.

examining how performance varies in different subsets of the data (to efficiently account for any post-randomization issue difficulty imbalances that may be more pronounced within subsets). See Appendix D for details of the regression approach used.

2.4 Supplementary Data Collection

Developer Forecasts and Post hoc Estimates Before randomizing each issue to either AI-allowed or AI-disallowed, we ask developers to forecast how long they expect each issue to take if they are allowed to use AI assistance, and if they are disallowed from using AI. This provides a measure of issue difficulty independent of random treatment assignment, and enables us to assess developer calibration and expectations regarding the usefulness of AI assistance for their work.

After the study's conclusion, we ask developers to estimate post hoc how much AI tooling affected the time it took them to complete issues. This helps us understand the relationship between individuals' beliefs about AI helpfulness and our measured effects.

The detailed instructions given to developers for per-issue forecasts and exit surveys are available in Section G.2.

Expert Forecasts We solicit forecasts from experts in machine learning and economics in order to compare our results against expert prior expectations regarding AI capabilities [36]. Machine learning experts are a mix of industry professionals from frontier AI labs, and academics from universities with leading graduate computer science programs. Economics experts are primarily academics and PhD students from universities with leading economics graduate programs. See Section E.5 for summary statistics of expert forecasts by expert group.

Experts are given detailed information about our study design, repositories participating in our experiment, and the proportion of developers who have prior experience with AI tooling (Section G.9). They then report their prediction of our point estimate for $\frac{\mathbb{E}[\text{time with AI disallowed}]}{\mathbb{E}[\text{time with AI allowed}]}$ [9].

To incentivize experts to make accurate forecasts, we offer to pay $\max(0, \$50 \times (1 - (\text{true answer} - \text{guessed answer})^2))$ for their point estimates.[10][11]

Cursor Analytics Data We collect per-developer analytics data from Cursor's analytics dashboard, for the 13 developers we provide Cursor Pro subscriptions to. For each developer, we collect their most-used model and the number of suggested and accepted lines of AI-generated code.

Exit Interviews and Surveys After the study's conclusion, we conduct exit interviews and surveys with all developers to assess where they found AI helpful, what strategies they used to effectively elicit work from AI, whether they felt they improved at using AI tooling over the course of the study, and to estimate how much they were sped up by AI during the study. Full details of the exit interviews are available in Section G.5.1.

Qualitative Evidence Throughout the study we collect qualitative evidence from developers, to form a more comprehensive understanding of their experiences using AI tools. Developers are instructed to take detailed notes regarding their experience and usage of AI tools, and we use inductive coding—where we iteratively read through the data to identify recurring patterns, create and refine categories as they emerge, and reorganize excerpts until stable themes develop—to cluster excerpts from these notes. Qualitative results in Section 3.3 (particularly quotes from developers) were collected primarily with this methodology. However, to investigate initial hypotheses, we often ask developers probing/targeted questions, so we cannot rule out bias from this type of qualitative evidence.

[9] Note that the estimate we use for Figure 1 is transformed to represent $\frac{\mathbb{E}[T \mid \text{AI=1}]}{\mathbb{E}[T \mid \text{AI=0}]} - 1$. It is not necessarily true that forecasters' belief about the point estimate of $\frac{\mathbb{E}[T \mid \text{AI=1}]}{\mathbb{E}[T \mid \text{AI=0}]}$ is equal to the reciprocal of their belief about the point estimate of $\frac{\mathbb{E}[T \mid \text{AI=0}]}{\mathbb{E}[T \mid \text{AI=1}]}$.

[10] Approximately one-third of forecasters are offered a maximum of \$100 rather than \$50.

[11] Taking a maximum with 0 makes our scoring rule improper [37]—their reward is not necessarily maximized at their true belief. We use this scoring rule for simplicity and clarity.

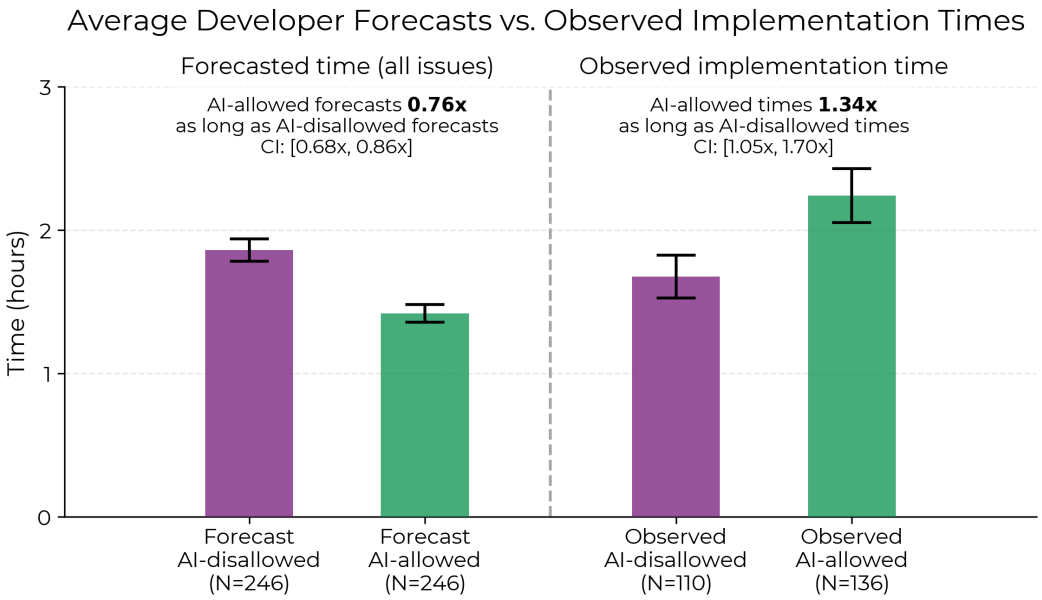


Figure 4: *Left: Raw average forecasted implementation times. Right: Raw average observed implementation times. The ratio of observed implementation times gives a more extreme slowdown estimate than regression-based estimates (Section D.1) because AI-allowed issues are forecasted (importantly, before treatment assignment) by developers to take slightly longer, which the regression corrects for. Both: Section D.4 describes confidence intervals around ratios of average times.*

Fine-Grained Screen Recording Activity Labels To compare how developers spend their time with and without AI assistance, we manually label a subset of 128 screen recordings with fine-grained activity labels, totaling 143 hours of video. In results based on these labeled screen recordings, we filter to remove issues where we find cheating, issues where the screen recording are broken for $>10\%$ of recording time, and issues with a $>20\%$ discrepancy between self-reported time and the recording time. This results in 74 recordings representing 84 hours of video for further analysis.

We label whether developers are: actively writing code, testing and debugging their code, reading or searching for information, using Git or managing their environment, prompting an AI system, waiting on an AI system to generate output, reviewing AI outputs, or idling/doing other miscellaneous work. Each high-level label is further broken down into one of 27 fine-grained categories. Labels have a resolution of ~ 10 seconds. Section G.8 describes the instructions and process used for labeling screen recordings.

3 Results

Developers complete 136 issues with AI-allowed and 110 issues with AI-disallowed. Section G.7 shows the number of issues completed across repositories and developers, respectively. We find that when developers use AI tools, they implement issues in 19% *more* time on average (Figure 1), and nearly all quantiles of observed implementation time see AI-allowed issues taking longer (Figure 5). That is, developers are slower when using AI is allowed. Colloquially, we refer to this result that issues with AI-allowed take longer than issues with AI-disallowed as *slowdown*.

3.1 Forecasts

Developer Forecasts and Post hoc Estimates Before developers complete each issue, they forecast how long they expect them to take with and without AI assistance. On average, they forecast speedup of 24%. Interestingly, after the experiment they post-hoc estimate that they were sped-up by 20% when using AI is allowed—after they used AI assistance, they estimate similar speedup as

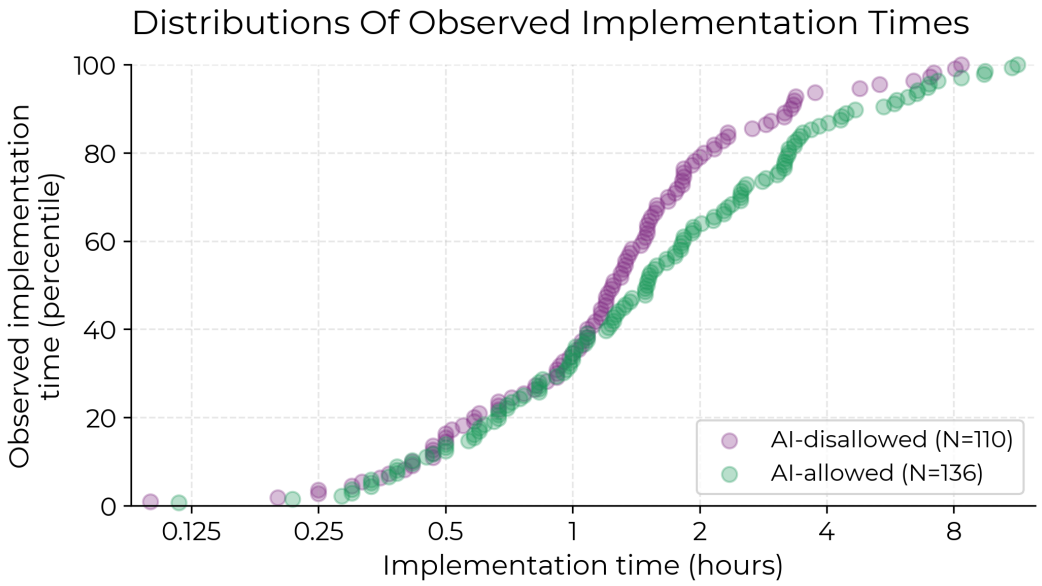


Figure 5: *Empirical cumulative distribution functions of observed implementation times. Percentile ordering is calculated separately for each treatment group.*

before, despite the fact that they are in fact slowed down by 19% (Figure 1). Figure 4 displays the raw average forecasted and observed implementation times[12].

Despite developers forecasting speedup from AI (while they are slowed down), developer forecasts *are* informative about completion time—the Pearson correlation between the time developers forecast AI-allowed issues taking and the actual time they take is 0.64, and the correlation between the time developers forecast AI-disallowed issues taking and the actual time time they take is 0.59. This suggests that developers are broadly well-calibrated on the relative amount of time that issues will take, but their expectations regarding the usefulness of AI assistance are reversed.

Expert Forecasts Speedup forecasts from 34 economics experts and 54 machine learning experts overestimate speedup even more drastically than developers, predicting AI will lead to decreases in implementation time of 39% and 38%, respectively (Figure 1). We show distributions of expert forecasts in Section E.5.

3.2 Activity Labels

On a subset of 74 issues for which we have valid screen recordings, we manually label the activities developers engage in while they work. Figure 6 shows the percentage of time developers spend for each type of issue (AI-allowed or AI-disallowed). When allowed to use AI, developers spend a smaller proportion of their time actively coding and reading/searching for information. Instead, they spend time reviewing AI outputs, prompting AI systems, and waiting for AI generations. Interestingly, they also spend a somewhat higher proportion of their time idle, where their screen recording doesn't show any activity. Section E.4 shows the number of minutes spent on average in each category (instead of percentage time spent), as well as the distributions of labels broken down into more fine-grained activities.

[12]The raw percentage difference in implementation times between AI-allowed and AI-disallowed issues is 34%, which is larger ini absolute value than the 19% slowdown estimated using the regression specified in Section D.1. This is true because AI-allowed issues ended up being slightly more difficult than AI-disallowed issues after randomization, as measured by the forecasted AI-disallowed times (not statistically significant; see Table 4). Our regression accounts for this, while this simple ratio estimator does not. See Figure 13 for results implied by alternative estimators.

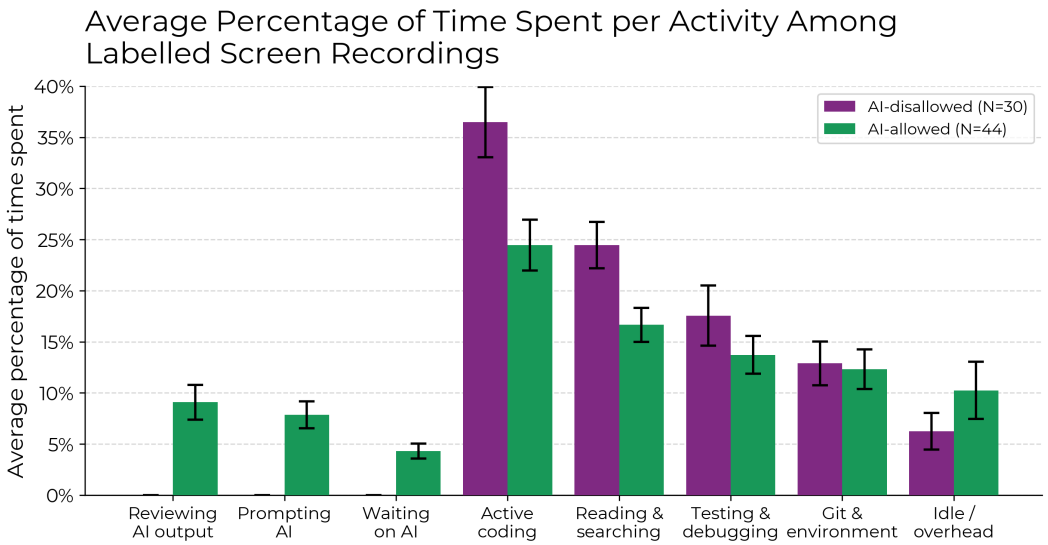


Figure 6: *On the subset of labeled screen recordings, when AI is allowed, developers spend less time actively coding and searching for/reading information, and instead spend time prompting AI, waiting on and reviewing AI outputs, and idle. Figure 18 shows the absolute (average) minutes spent in each category, and Figure 20 presents these results broken down into 27 fine-grained categories.*

3.3 Factor Analysis

Given the surprising nature of this result, we investigate 20 potential contributing factors that may contribute to developers spending more time on tasks when AI usage is allowed. We group these factors into four categories:

- **Direct productivity loss** (⏳): mechanisms by which the use of AI tools actively slows down development.
- **Experimental artifact** (🧪): confounders from our experimental setup or procedures that may introduce biases, or limit the external validity.
- **Raises developer performance** (👤): attributes of the issues, repositories, or setting that improve developer ability relative to AI.
- **Limits AI performance** (🤖): attributes of the issues, repositories, or AI/environment tooling that diminish AI's effectiveness relative to developers.

Using entry and exit surveys, screen recordings, developer interviews, and subset analyses we find qualitative and quantitative evidence that 5 of the 20 factors contribute to slowdown, we find mixed/unclear/no evidence that 9 of the factors contribute to slowdown, and we find evidence against 6 of the factors contributing. However, we strongly caution against over-indexing on the basis of any individual pieces of evidence, as we are not powered for statistically significant multiple comparisons when subsetting our data. This analysis is intended to provide speculative, suggestive evidence about the mechanisms behind slowdown. Appendix C discusses the evidence for/against each factor in Table 1.

4 Discussion

We provide evidence that recent AI systems slow down experienced open-source developers with moderate AI experience completing real issues on large, popular repositories they are highly familiar with. This observed slowdown serves as some evidence that AI capabilities in the wild may be lower than results on commonly used benchmarks may suggest.

Furthermore, we show that both experts and developers drastically overestimate the usefulness of AI on developer productivity, *even after they have spent many hours using the tools*. This underscores

Factors likely to contribute to slowdown

Factor	Type	Relevant Observations
Over-optimism about AI usefulness (C.1.1)	⏳	• Developers forecast AI will decrease implementation time by 24% • Developers post hoc estimate AI decreased implementation time by 20%
High developer familiarity with repositories (C.1.2)	👤	• Developers slowed down more on issues they are more familiar with • Developers report that their experience makes it difficult for AI to help them • Developers average 5 years experience and 1,500 commits on repositories
Large and complex repositories (C.1.3)	🤖	• Developers report AI performs worse in large and complex environments • Repositories average 10 years old with >1,100,000 lines of code
Low AI reliability (C.1.4)	🤖	• Developers accept <44% of AI generations • Majority report making major changes to clean up AI code • 9% of time spent reviewing/cleaning AI outputs
Implicit repository context (C.1.5)	🤖👤	• Developers report AI doesn't utilize important tacit knowledge or context

Factors with unclear effect on slowdown

Factor	Type	Relevant Observations
Experimentally driven overuse of AI (C.2.1)	🧪	• Developers sometimes report overuse due to experiment • Similar slowdown from developers reporting overuse vs. normal use
Unrepresentative task distribution (C.2.2)	🧪	• Developers report issues are standard but on the shorter side • Excludes non-programming tasks developers complete in normal work
AI increasing issue scope (C.2.3)	🧪	• Developers who report scope creep see *less* slowdown • Mixed developer reports on AI's impact on scope • 47% more lines of code per forecasted hour in AI-allowed issues
Bias from issue completion order (C.2.4)	🧪	• Developers decide order post randomization
Trading speed for ease (C.2.5)	⏳	• Some developers report using AI is less effortful • High developer retention on Cursor
Low quality initial pull requests (C.2.6)	⏳	• Minor difference in mean post-review times between conditions • Qualitatively similar PR quality between conditions
Below-average use of AI tools (C.2.7)	🤖	• Similar slowdown for developers with prior Cursor experience • No clear learning effect across first 30-50 hours of Cursor usage • Developers appear qualitatively in distribution for Cursor Pro users
AI generation latency (C.2.8)	🤖	• Mixed developer reports that waiting on AI generations was important • Developers spend 4% of time waiting on AI generations
Suboptimal elicitation (C.2.9)	🤖	• Developers use Cursor agents/chat in majority of AI-allowed issues • Developers sample few tokens from models • But existing literature finding positive speedup also uses few tokens • Unused elicitation strategies could improve AI reliability

Factors unlikely to contribute to slowdown

Factor	Type	Relevant Observations
Unfamiliar development environment (C.3.1)	🧪	• Most developers use comparable IDEs between treatment conditions • These developers still see slowdown of 24% • No clear learning effects across first 30-50 hours of Cursor usage
Cheating or under-use of AI (C.3.2)	🧪	• AI used in all but 16.4% of allowed cases with labeled screen recordings • Only 3 cheating instances in 54 screen recordings
Issue dropout (C.3.3)	🧪	• Developers with no accidental dropout see similar slowdown • Issues dropped intentionally are qualitatively unbiased
Non-robust outcome measure (C.3.4)	🧪	• Alternative outcome measures yield similar slowdown
Non-robust estimator (C.3.5)	🧪	• Alternative estimators yield similar slowdown
Non-frontier model usage (C.3.6)	🤖	• Developers mostly use (at the time) frontier models

Table 1: *Summary of factors that may* a priori *explain or contribute to slowdown, grouped by the state of evidence for or against their impact on the slowdown effect.* 👤 *are factors that raise human performance,* 🤖 *are factors that limit AI performance,* 🧪 *are experimental artifacts that may bias/confound results, and* ⏳ *are factors that directly contribute to productivity losses.*

the importance of conducting field experiments with robust outcome measures, compared to relying solely on expert forecasts or developer surveys.

4.1 Key Caveats

Setting-specific factors We caution readers against overgeneralizing on the basis of our results. The slowdown we observe does *not* imply that current AI tools do not often improve developer's productivity—we find evidence that the high developer familiarity with repositories and the size and maturity of the repositories both contribute to the observed slowdown, and these factors do not apply in many software development settings. For example, our results are consistent with small greenfield projects or development in unfamiliar codebases seeing substantial speedup from AI assistance.

AI-specific factors We expect that AI systems that have higher fundamental reliability, lower latency, and/or are better elicited (e.g. via more inference compute/tokens, more skilled prompting/scaffolding, or explicit fine-tuning on repositories) could speed up developers in our setting (i.e. experienced open-source developers on large repositories).

Agents can make meaningful progress on issues We have preliminary evidence (forthcoming) that fully autonomous AI agents using Claude 3.7 Sonnet can often correctly implement the core functionality of issues on several repositories that are included in our study, although they fail to fully satisfy all requirements (typically leaving out important documentation, failing linting/styling rules, and leaving out key unit or integration tests). This represents immense progress relative to the state of AI just 1-2 years ago, and if progress continues apace (which is *a priori* at least plausible, although not guaranteed), we may soon see significant speedup in this setting.

5 Acknowledgments

We thank the open-source developers who participated in this study. Your hard work, diligent record keeping, and excellent software made it a pleasure to work with you. Thanks to Aaron Diamond-Reivich, Alan Akbik, Domenic Denicola, Dens Sumesh, Jaden Fiotto-Kaufman, João Gante, Liam DeVoe, Matthew Pickering, Muhammad Haris, Philipp Burckhardt, Quentin Anthony, Ruben Bloom, Sam Derbyshire, and other participating developers.

We thank the following reviewers for feedback on the experimental design and paper drafts: Adrien Ecoffet, Alexander Barry, Ali Merali, Ajeya Cotra, Andres Campero, Andrey Fradkin, Basil Halperin, Cozmin Ududec, Eli Lifland, Ernest Davis, Gregory Sun, Hjalmar Wijk, James Requeima, Jide Alaga, Josh Jacobson, Lawrence Chan, Megan Kinniment, Michael Sklar, Neev Parikh, Rif A. Saurous, Rob Miles, Ryan Greenblatt, Seraphina Nix, Sydney Von Arx, Thomas Kwa, and Tom Cunningham.

We thank the following individuals for help with data collection: Adam Hanson, Amy Ngo, Chris Canal, Jebastin Nadar, Luis Slyfield, and Martin Milbradt.

We thank the Sami Jawar, Thomas Broadley for technical support throughout the project.

We thank the following for their operational support through the project: Bhaskar Chaturvedi, Emma Abele, Kit Harris, Kris Chari, Kyle Scott, Rebecca Baron, and Rae She.

The authors thank Stephanie He for graphic design contributions.

The authors especially thank Aron Lajko, Chris Painter, Jasmine Dhaliwal, and Steve Newman for close review, feedback, and support throughout the project.

References

[1] Nestor Maslej, Loredana Fattorini, Raymond Perrault, Yolanda Gil, Vanessa Parli, Njenga Kariuki, Emily Capstick, Anka Reuel, Erik Brynjolfsson, John Etchemendy, Katrina Ligett, Terah Lyons, James Manyika, Juan Carlos Niebles, Yoav Shoham, Russell Wald, Tobi Walsh, Armin Hamrah, Lapo Santarlasci, Julia Betts Lotufo, Alexandra Rome, Andrew Shi, and Sukrut Oak. Artificial intelligence index report 2025, 2025. URL https://arxiv.org/abs/2504.07139.

[2] Thomas Kwa, Ben West, Joel Becker, Amy Deng, Katharyn Garcia, Max Hasin, Sami Jawhar, Megan Kinniment, Nate Rush, Sydney Von Arx, Ryan Bloom, Thomas Broadley, Haoxing Du, Brian Goodrich, Nikola Jurkovic, Luke Harold Miles, Seraphina Nix, Tao Lin, Neev Parikh, David Rein, Lucas Jun Koba Sato, Hjalmar Wijk, Daniel M. Ziegler, Elizabeth Barnes, and Lawrence Chan. Measuring ai ability to complete long tasks, 2025. URL https://arxiv.org/abs/2503.14499.

[3] Hjalmar Wijk, Tao Lin, Joel Becker, Sami Jawhar, Neev Parikh, Thomas Broadley, Lawrence Chan, Michael Chen, Josh Clymer, Jai Dhyani, Elena Ericheva, Katharyn Garcia, Brian Goodrich, Nikola Jurkovic, Holden Karnofsky, Megan Kinniment, Aron Lajko, Seraphina Nix, Lucas Sato, William Saunders, Maksym Taran, Ben West, and Elizabeth Barnes. Re-bench: Evaluating frontier ai r&d capabilities of language model agents against human experts, 2025. URL https://arxiv.org/abs/2411.15114.

[4] Jun Shern Chan, Neil Chowdhury, Oliver Jaffe, James Aung, Dane Sherburn, Evan Mays, Giulio Starace, Kevin Liu, Leon Maksin, Tejal Patwardhan, Lilian Weng, and Aleksander Madry. Mle-bench: Evaluating machine learning agents on machine learning engineering, 2025. URL https://arxiv.org/abs/2410.07095.

[5] Giulio Starace, Oliver Jaffe, Dane Sherburn, James Aung, Jun Shern Chan, Leon Maksin, Rachel Dias, Evan Mays, Benjamin Kinsella, Wyatt Thompson, Johannes Heidecke, Amelia Glaese, and Tejal Patwardhan. Paperbench: Evaluating ai's ability to replicate ai research, 2025. URL https://arxiv.org/abs/2504.01848.

[6] Shanghaoran Quan, Jiaxi Yang, Bowen Yu, Bo Zheng, Dayiheng Liu, An Yang, Xuancheng Ren, Bofei Gao, Yibo Miao, Yunlong Feng, Zekun Wang, Jian Yang, Zeyu Cui, Yang Fan, Yichang Zhang, Binyuan Hui, and Junyang Lin. Codeelo: Benchmarking competition-level code generation of llms with human-comparable elo ratings, 2025. URL https://arxiv.org/abs/2501.01257.

[7] Samuel Miserendino, Michele Wang, Tejal Patwardhan, and Johannes Heidecke. Swe-lancer: Can frontier llms earn $1 million from real-world freelance software engineering?, 2025. URL https://arxiv.org/abs/2502.12115.

[8] David Rein, Betty Li Hou, Asa Cooper Stickland, Jackson Petty, Richard Yuanzhe Pang, Julien Dirani, Julian Michael, and Samuel R. Bowman. Gpqa: A graduate-level google-proof q&a benchmark, 2023. URL https://arxiv.org/abs/2311.12022.

[9] David Rein, Joel Becker, Amy Deng, Seraphina Nix, Chris Canal, Daniel O'Connel, Pip Arnott, Ryan Bloom, Thomas Broadley, Katharyn Garcia, Brian Goodrich, Max Hasin, Sami Jawhar, Megan Kinniment, Thomas Kwa, Aron Lajko, Nate Rush, Lucas Jun Koba Sato, Sydney Von Arx, Ben West, Lawrence Chan, and Elizabeth Barnes. Hcast: Human-calibrated autonomy software tasks, 2025. URL https://arxiv.org/abs/2503.17354.

[10] Sida Peng, Eirini Kalliamvakou, Peter Cihon, and Mert Demirer. The impact of ai on developer productivity: Evidence from github copilot, 2023. URL https://arxiv.org/abs/2302.06590.

[11] Elise Paradis, Kate Grey, Quinn Madison, Daye Nam, Andrew Macvean, Vahid Meimand, Nan Zhang, Ben Ferrari-Church, and Satish Chandra. How much does ai impact development speed? an enterprise-based randomized controlled trial, 2024. URL https://arxiv.org/abs/2410.12944.

[12] Inioluwa Deborah Raji, Emily M. Bender, Amandalynne Paullada, Emily Denton, and Alex Hanna. AI and the everything in the whole wide world benchmark. *CoRR*, abs/2111.15366, 2021. URL https://arxiv.org/abs/2111.15366.

[13] Stella Biderman, Hailey Schoelkopf, Lintang Sutawika, Leo Gao, Jonathan Tow, Baber Abbasi, Alham Fikri Aji, Pawan Sasanka Ammanamanchi, Sidney Black, Jordan Clive, Anthony DiPofi, Julen Etxaniz, Benjamin Fattori, Jessica Zosa Forde, Charles Foster, Jeffrey Hsu, Mimansa Jaiswal, Wilson Y. Lee, Haonan Li, Charles Lovering, Niklas Muennighoff, Ellie Pavlick, Jason Phang, Aviya Skowron, Samson Tan, Xiangru Tang, Kevin A. Wang, Genta Indra Winata, François Yvon, and Andy Zou. Lessons from the trenches on reproducible evaluation of language models, 2024. URL https://arxiv.org/abs/2405.14782.

[14] Ernest Davis. Benchmarks for automated commonsense reasoning: A survey, 2023. URL https://arxiv.org/abs/2302.04752.

[15] Leonardo Gambacorta, Han Qiu, Shuo Shan, and Daniel M Rees. *Generative AI and labour productivity: a field experiment on coding*, volume 1208. Bank for International Settlements, Monetary and Economic Department, 2024.

[16] Doron Yeverechyahu, Raveesh Mayya, and Gal Oestreicher-Singer. The impact of large language models on open-source innovation: Evidence from github copilot, 2025. URL https://ssrn.com/abstract=4684662.

[17] Zheyuan Cui, Mert Demirer, Sonia Jaffe, Leon Musolff, Sida Peng, and Tobias Salz. The effects of generative ai on high-skilled work: Evidence from three field experiments with software developers, June 2025. URL https://ssrn.com/abstract=4945566.

[18] Thomas Weber, Maximilian Brandmaier, Albrecht Schmidt, and Sven Mayer. Significant productivity gains through programming with large language models. *Proc. ACM Hum.-Comput. Interact.*, 8(EICS), June 2024. doi: 10.1145/3661145. URL https://doi.org/10.1145/3661145.

[19] OpenAI. OpenAI o3 and o4-mini System Card. https://cdn.openai.com/pdf/2221c875-02dc-4789-800b-e7758f3722c1/o3-and-o4-mini-system-card.pdf, 2025. [Accessed 23-06-2025].

[20] Anthropic. Anthropic Claude 4 System Card. https://www-cdn.anthropic.com/6be99a52cb68eb70eb9572b4cafad13df32ed995.pdf, 2025. [Accessed 23-06-2025].

[21] Ajay Agrawal, Joshua S. Gans, and Avi Goldfarb. Artificial intelligence: The ambiguous labor market impact of automating prediction. *Journal of Economic Perspectives*, 33(2):31–50, May 2019. doi: 10.1257/jep.33.2.31. URL https://www.aeaweb.org/articles?id=10.1257/jep.33.2.31.

[22] Erik Brynjolfsson, Danielle Li, and Lindsey Raymond. Generative ai at work*. *The Quarterly Journal of Economics*, 140(2):889–942, 02 2025. ISSN 0033-5533. doi: 10.1093/qje/qjae044. URL https://doi.org/10.1093/qje/qjae044.

[23] Shakked Noy and Whitney Zhang. Experimental evidence on the productivity effects of generative artificial intelligence. *Science*, 381(6654):187–192, 2023. doi: 10.1126/science.adh2586. URL https://www.science.org/doi/abs/10.1126/science.adh2586.

[24] Jonathan H. Choi and Daniel Schwarcz. Ai assistance in legal analysis: An empirical study. *Journal of Legal Education*, 73(2), 2025. URL https://jle.aals.org/home/vol73/iss2/5/.

[25] Anthropic. Responsible scaling policy evaluations report – claude 3 opus. Technical report, Anthropic, 2024. URL https://cdn.sanity.io/files/4zrzovbb/website/210523b8e11b09c704c5e185fd362fe9e648d457.pdf. Accessed June 2025.

[26] C. Mouton, Caleb Lucas, and Ella Guest. The operational risks of ai in large-scale biological attacks. Research report, RAND Corporation, Santa Monica, 2024. URL https://www.rand.org/content/dam/rand/pubs/research_reports/RRA2900/RRA2977-2/RAND_RRA2977-2.pdf.

[27] Aaron Grattafiori et al. The llama 3 herd of models, 2024. URL https://arxiv.org/abs/2407.21783.

[28] Tejal Patwardhan, Kevin Liu, Todor Markov, Neil Chowdhury, Dillon Leet, Natalie Cone, Caitlin Maltbie, Joost Huizinga, Carroll Wainwright, Shawn (Froggi) Jackson, Steven Adler, Rocco Casagrande, and Aleksander Madry. Building an early warning system for LLM-aided biological threat creation. https://openai.com/index/building-an-early-warning-system-for-llm-aided-biological-threat-creation/, January 2024. Accessed: 2025-06-25.

[29] Jared Leibowich, Nikola Jurkovic, and Tom Davidson. Could advanced ai accelerate the pace of ai progress? interviews with ai researchers, 2024. URL https://ssrn.com/abstract=5115692.

[30] Ege Erdil and Tamay Besiroglu. Explosive growth from ai automation: A review of the arguments, 2024. URL https://arxiv.org/abs/2309.11690.

[31] Ege Erdil, Andrei Potlogea, Tamay Besiroglu, Edu Roldan, Anson Ho, Jaime Sevilla, Matthew Barnett, Matej Vrzla, and Robert Sandler. Gate: An integrated assessment model for ai automation, 2025. URL https://arxiv.org/abs/2503.04941.

[32] Tom Davidson. What a compute-centric framework says about takeoff speeds. https://www.openphilanthropy.org/research/what-a-compute-centric-framework-says-about-takeoff-speeds/, June 2023. Open Philanthropy. Accessed: 2025-06-25.

[33] Daron Acemoglu. The simple macroeconomics of ai. *Economic Policy*, 40(121):13–58, 08 2024. ISSN 0266-4658. doi: 10.1093/epolic/eiae042. URL https://doi.org/10.1093/epolic/eiae042.

[34] Ajay Agrawal, Joshua Gans, and Avi Goldfarb. Economic policy for artificial intelligence. *Innovation Policy and the Economy*, 19:139–159, 2019. doi: 10.1086/699935. URL https://doi.org/10.1086/699935.

[35] Jason Furman and Robert Seamans. *AI and the Economy*, pages 161–191. University of Chicago Press, May 2018. doi: 10.1086/699936. URL http://www.nber.org/chapters/c14099.

[36] Stefano DellaVigna, Nicholas Otis, and Eva Vivalt. Forecasting the results of experiments: Piloting an elicitation strategy. *AEA Papers and Proceedings*, 110:75–79, May 2020. doi: 10.1257/pandp.20201080. URL https://www.aeaweb.org/articles?id=10.1257/pandp.20201080.

[37] Tilmann Gneiting and Adrian E Raftery. Strictly proper scoring rules, prediction, and estimation. *Journal of the American Statistical Association*, 102(477):359–378, 2007. doi: 10.1198/016214506000001437. URL https://doi.org/10.1198/016214506000001437.

[38] Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. SWE-bench: Can language models resolve real-world github issues? In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=VTF8yNQM66.

[39] Nelson F. Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. Lost in the middle: How language models use long contexts, 2023. URL https://arxiv.org/abs/2307.03172.

[40] Charlie Snell, Jaehoon Lee, Kelvin Xu, and Aviral Kumar. Scaling llm test-time compute optimally can be more effective than scaling model parameters, 2024. URL https://arxiv.org/abs/2408.03314.

[41] Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. Self-consistency improves chain of thought reasoning in language models, 2023. URL https://arxiv.org/abs/2203.11171.

[42] Stack Overflow. 2024 developer survey: Technology - integrated development environment, 2024. URL https://survey.stackoverflow.co/2024/technology#1-integrated-development-environment. Accessed: June 26, 2025.

A Author contributions

Joel Becker and **Nate Rush** designed, implemented, and led the project.

Beth Barnes gave feedback and guidance on the project.

David Rein contributed substantially to the writing and framing of the results.

B Extended Discussion

We do *not* provide evidence that:	**Clarification**
AI systems do not currently speed up many or most software developers	We do not claim that our developers or repositories represent a majority or plurality of software development work
AI systems do not speed up individuals or groups in domains other than software development	We only study software development
AI systems in the near future will not speed up developers in our exact setting	Progress is difficult to predict, and there has been substantial AI progress over the past five years [2]
There are not ways of using existing AI systems more effectively to achieve positive speedup in our exact setting	Cursor does not sample many tokens from LLMs, it may not use optimal prompting/scaffolding, and domain/repository-specific training/finetuning/few-shot learning could yield positive speedup

Table 2: *Potential misconceptions about our work: what our evidence does not demonstrate about AI and developer productivity.*

Potential Misreadings of Results Given both the importance of understanding AI capabilities/risks, and the diversity of perspectives on these topics, we feel it's important to forestall potential misunderstandings or over-generalizations of our results. We list claims that we do ***not*** provide evidence for in Table 2.

Paper	**Result**	**AI $\geq$ GPT-4?**	**Non-synthetic tasks**	**Experienced, high-familiarity devs**	**Fixed outcome measure**
Peng et al. [10]	$\uparrow$ 56% faster	✗	✗	✗	✓
Weber et al. [18]	$\uparrow$ 65% faster	✗	✗	✗	✓
Cui et al. [17]	$\uparrow$ 26% output	✗	✓	✓	✗
Paradis et al. [11]	$\uparrow$ 21% faster	**?**	✗	✗	✓
Gambacorta et al. [15]	$\uparrow$ 55% output	✗	✓	✓	✗
Yeverechyahu et al. [16]	$\uparrow$ 37% output	✗	✓	✓	✗
Our study	$\downarrow$ **19% slower**	✓	✓	✓	✓

Table 3: *Overview of key studies measuring the impact of AI tools on software development productivity. Paradis et al. [11] does not report the model(s) used internally.*

Literature Comparison Table 3 compares relevant studies that measure the impact of AI tools on software developer productivity, along key dimensions that distinguish our results from prior work.

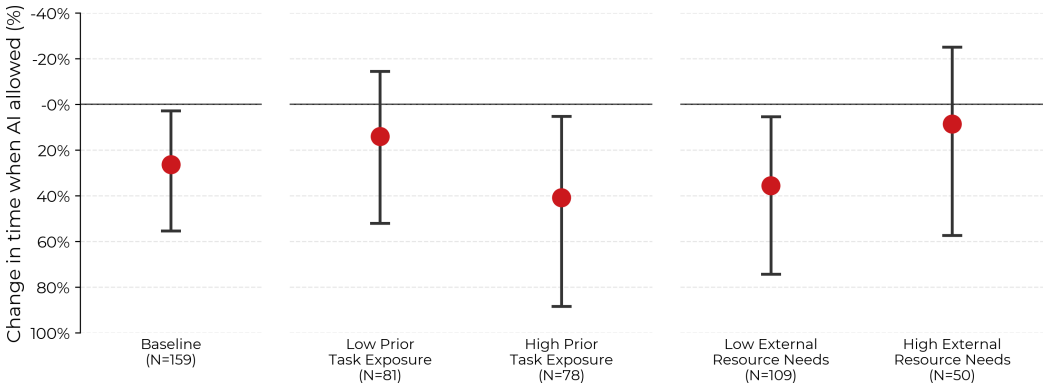


Figure 7: *Developers are slowed down more on issues where they self-report having significant prior task exposure, and on issues where they self-report having low external resource needs (e.g. documentation, reference materials). We only collected this data for the latter half of issues completed in the study—this is why we have a smaller number of issues for our baseline slowdown estimate. See Section D.3 for details on how we estimate heterogeneous treatment effects.*

Other relevant literature typically does not investigate the impact of AI systems more capable than GPT-4 (which as of mid-2025 isn't itself close to the frontier), and does not (in any individual study) analyze tasks in the wild, with experienced developers, using outcome measures that correspond directly with productivity and that are fixed before treatment assignment.

C Factor Analysis

C.1 Factors driving slowdown

We observe 5 setting-specific factors that contribute to our observed slowdown, summarized in Table 1.

C.1.1 Over-optimism about AI usefulness *(Direct productivity loss)*

On AI-allowed issues, developers are not *required* to use AI tools—they are instructed to use AI assistance as much or as little as they would find helpful. Given this, we might expect the percentage change in issue completion time due to AI to be lower-bounded at 0%—if a developer is aiming to be as productive as possible, and they see that they are being slowed down by an AI tool, we'd expect them to not continue using the AI tool.

However, developers have an overoptimistic picture of how AI affects their productivity, both before and after they complete issues. Before starting issues, developers forecast that using AI will reduce completion time by 24%. After completing issues, developers estimate that using AI reduced their issue completion time by 20% on average. This overoptimistic view may lead developers to overuse AI assistance, despite its negative effect on their productivity.

C.1.2 High developer familiarity with repositories *(Raises developer performance)*

We might expect that AI assistance is less helpful on tasks where developers are already highly skilled, for example, because they have completed similar tasks previously, or because they have all of the knowledge and skills required already.

Before developers complete each issue, they score how much "Prior Task Exposure", they have from 1 - 5, where 1 means "Never done this type of task before", and 5 means "I am an expert at this type of task". We consider ratings of less than or equal to 3 to mean they have low prior task exposure, and ratings higher than 3 to imply high prior task exposure (we selected this cutoff to achieve roughly balanced ratings).

Figure 7 gives moderate evidence that developers are slowed down more on on issues where they have high prior task exposure.

Developers also rate (before issues are randomized and completed) "External Resource Needs" from 1 - 3, where 1 means "I need extensive documentation / reference material / googling to complete this task" and 3 means "I could complete this task entirely from memory and experience." We consider ratings of less than or equal to 2 to mean they have low external resource needs, and ratings higher than 2 to imply high external resource needs (we selected this cutoff to achieve roughly balanced ratings). Figure 7 presents moderate evidence that developers are slowed down more on issues where they need fewer external resources.

Qualitatively, developers note that AI is particularly helpful when working on unfamiliar issues, and less helpful when working on familiar ones.

One developer working with unfamiliar datasets found that AI was helpful in answering *"general questions about e.g. EICAR."* Another developer noted that Cursor was *"super helpful in figuring out how to write a [frontend test.] I didn't know how to do this before and on my third time asking cursor for help with it, it came up with this solution."* Another developer, working with Git hooks, noted that *"Given that it was my first time with Git hooks, without AI the implementation would've taken me [3 additional hours]."* Sometimes, portions of one's own codebase can be as unknown as a new API. One developer noted that *"cursor found a helper test function that I didn't even know existed when I asked it how we tested deprecations."*

On the other hand, developers note that AI is much less helpful on issues where they are expert. One developer notes that *"if I am the dedicated maintainer of a very specialized part of the codebase, there is no way agent mode can do better than me."*

Broadly, we present moderate evidence that on the issues in our study, developers are slowed down more when they have high prior task exposure and lower external resource needs. We hypothesize that analogously, AI helps our developers less compared to existing literature [10; 11] because our developers have substantially more experience on their respective repositories (5 years and 1,500 commits on average). This would be consistent with the experience/familiarity effects observed in Noy and Zhang [23]; Cui et al. [17].

C.1.3 Large and complex repositories *(Limits AI performance)*

Developers qualitatively note LLM tooling performs worse in more complex environments. One developer says *"it also made some weird changes in other parts of the code that cost me time to find and remove [...] My feeling is the refactoring necessary for this PR was "too big" [and genAI] introduced as many errors as it fixed."* Another developer comments that one prompt *"failed to properly apply the edits and started editing random other parts of the file,"* and that these failures seemed to be heavily related to *"the size of a single file it is attempting to perform edits on."*.

We hypothesize that analogously to these size and complexity effects within our study, AI broadly helps our developers less compared to existing randomized controlled trials (RCTs) measuring speedup from AI tools because of the overall size and complexity of the repositories included—participating repositories are on average about 10 years old and contain >1,100,000 lines of code, compared to the more greenfield projects completed in Peng et al. [10], Paradis et al. [11], and Weber et al. [18]. This would be consistent with existing literature studying the effects of environment complexity on AI performance [3; 38; 39].

C.1.4 Low AI reliability *(Limits AI performance)*

When using Cursor, developers accept <44% of the generations.[13] When developers do not accept generations, we observe a mix of reattempting with different prompts, and giving up (i.e. reverting the proposed changes).

This relatively low reliability qualitatively results in significant wasted time, as developers often spend time reviewing, testing, or modifying AI generated code before they decide to reject it. One developer notes that he *"wasted at least an hour first trying to [solve a specific issue] with AI"* before eventually reverting all code changes and just implementing it without AI assistance.

[13] We were not able to collect this data from 3 developers who used their own pre-existing Cursor Pro subscriptions, so this statistic excludes them.

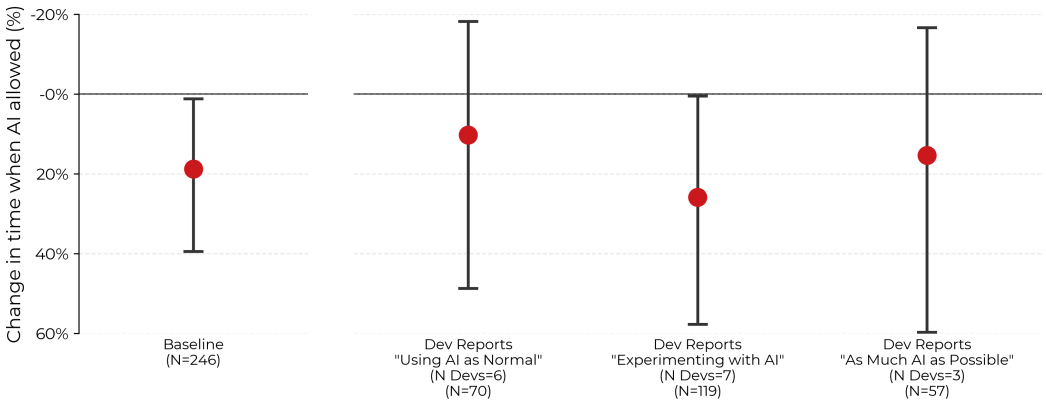


Figure 8: *Developers who report that they were experimenting with AI or using AI as much as possible see greater slowdown than developers who report using AI as they normally would. See Section D.3 for details on how we estimate heterogeneous treatment effects.*

Developers further note that even when they accept AI generations, they spend a significant amount of time reviewing and editing AI generated code to ensure it meets their high standards. 75% report that they read every line of AI generated code, and 56% of developers report that they often need to make major changes to clean up AI code—when asked, 100% developers report needing to modify AI generated code. One developer comments that AI generated code resulted in *"more to fix up, in terms of code conventions, code style"*. Another developer says *"I accept [AI generated code] [...] then I feel like I do a lot of simplifying its code."*

Quantitatively, on the subset of 44 issues with valid labeled screen recordings, we find that when AI is allowed, developers spend approximately 9% of their time reviewing and cleaning AI generated outputs when working with AI.

C.1.5 Implicit repository context *(Limits AI performance, Raises developer performance)*

In software development, developers often rely on their own undocumented knowledge of the codebase to assist design and implementation decisions. In our study, developers often note that AIs lack this tacit codebase knowledge, resulting in less useful AI outputs. One developer notes that AI often acts like a new contributor to the repository, and that *"AI doesn't pick the right location to make the edits."* Another developer notes that while *"we [..] know the data that will interact with the code, but the model doesn't know the data. It doesn't know we need to take care of this weird case of backwards compatibility and [thus] keep this specific line. And this is very hard to give as [context to the model]."*.

We hypothesize that the size and maturity of the included repositories increases the amount of tacit knowledge that experienced developers rely on when completing their work—because AI systems may have less access to this knowledge, it may be more difficult for them to assist experienced developers on these issues.

C.2 Factors with an unclear effect on slowdown

We consider 9 factors that have mixed/unclear directional effect on speedup, including effects for which we have no evidence in either direction.

C.2.1 Experimentally driven overuse of AI *(Experimental artifact)*

As this study is not blinded, developers' awareness of the study may have changed their behavior.

Developers were instructed to use AI to whatever degree they thought would make them most productive. After the study's conclusion, we ask developers which best describes their AI usage: "using AI as normal", "experimenting with AI" or "using as much AI as possible." Figure 8 doesn't show a clear effect when estimating speedup broken down by these labels.

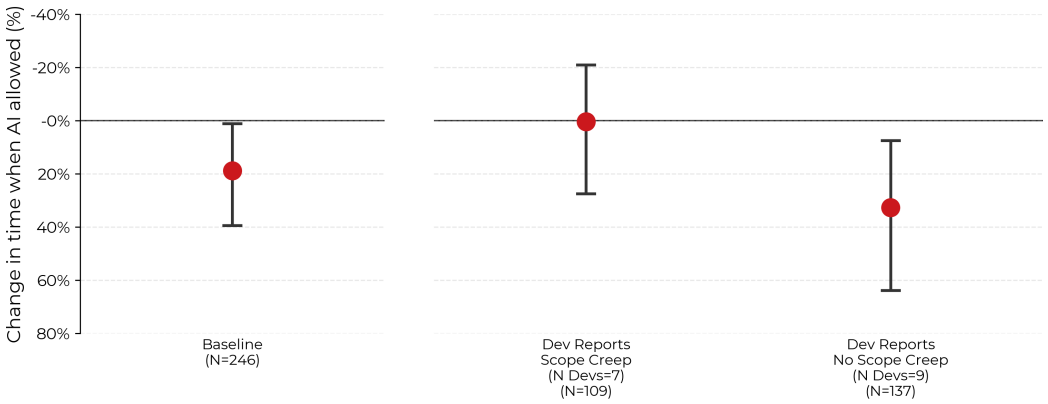


Figure 9: *Speedup on issues broken down by whether the developer completing the issue reports scope creep when using AI. See Section D.3 for details on how we estimate heterogeneous treatment effects.*

However, qualitatively, several developers reported that their participation in the experiment sometimes led to them overusing AI in ways that were unproductive.

C.2.2 Unrepresentative task distribution *(Experimental artifact)*

The issues are intended to be as similar as possible to those that would have been worked on if this study never took place. Developers who completed the study noted that issues were *"completely standard"* and that *"there's nothing I did for this that I wouldn't have done otherwise."*, and manual review of issues confirm that the issues represent normal work on each repository.

However, the issues completed during this study do not capture all contributions developers make to these repositories. As developers were asked to break larger issues into ≤ 2 hour issues if possible, developers note that issues were smaller than average than their normal work. One developer comments *"It was the same set of tasks, but sampled from the small end."* Furthermore, developers make other types of contributions to these repositories—the work of resolving issues does not capture PR review, or design discussions, for example.

These selection pressures may have biased issues to be better scoped and more clearly defined than the average work that developers complete on repositories. Existing literature would suggest that this means AIs would perform better on these issues, which we might expect to speed up developers (in contrast to our observed slowdown). However, better scoped issues may also be issues where these expert developers perform better as well, making AIs less relatively useful, so the net effect of this selection is unclear.

C.2.3 AI increasing issue scope *(Experimental artifact)*

A key design decision for our study is that issues are defined *before* they are randomized to AI-allowed or AI-disallowed groups, which helps avoid confounding effects on the outcome measure (in our case, the time issues take to complete). However, issues vary in how precisely their scope is defined, so developers often have some flexibility with what they implement for each issue. This raises a concern for measuring the impact of AI assistance on developer productivity—if developers expand the scope of their work when using AI tools, even if those issues take longer their productivity might be similar to AI-disallowed issues (because they are getting more done).

We survey developers after the study period has ended, and ask if on average, they believed that they experienced scope creep when working on AI-allowed issues. Figure 9 shows percentage change in issue completion time due to AI broken down by responses to this question—interestingly, we see that developers who report scope creep on AI-allowed issues are actually slowed down *less* than developers who don't report experiencing scope creep. This is evidence against the hypothesis that slowdown is caused by developers increasing the scope of AI-allowed issues, however, it relies on developer self-reports of scope creep, which may be unreliable or subjective.

Qualitatively, some developers self-report that AI does not change their implementations. One developer notes that due to his experience and knowing how changes should be made, *"the scope is not variable."* Another developer notes that AI would not change his approach to an issue, as *"AI was much more suggestable than I was."*

On the other hand, some developers note that the scope of issues sometimes changes as a result of AI. One developer comments *"If I didn't have AI, I probably [...] not gone so ham on nailing the correct Python tooling."* Another developer notes *"some of [code changes] were a little tedious, and so I am like is this going to [be worth it without AI], but with AI [I make the changes]."*

Quantitatively, we observe that developers write 47% ($p = 0.16$) more code per hour of AI-disallowed forecasts, on issues where AI is allowed. This is not statistically significant, and as discussed in section 1, an increase in the number of lines of code does not necessarily correspond to greater productivity or a larger scope, because code can be more verbose but functionally equivalent. Furthermore, this metric contains additional noise because files can be programatically generated by automated systems, and we're unable to collect the exact lines of code written by the human (or human plus AI).

C.2.4 Bias from issue completion order *(Experimental artifact)*

Each developer typically has several issues that are randomized in batches, and can then choose the order they complete these issues. This could cause a bias in completion times. For example, if developers prefer to complete AI-disallowed issues when they have more energy, they may choose to do do AI-disallowed issues first more often than AI-allowed issues, which could cause AI-allowed issues to take longer even if AI would have decreased the implementation time without this ordering effect.

While we receive no qualitative reports from developers that they prioritize issues in this way, we do not have access to developers entire workflows, so we cannot fully rule out this effect.

C.2.5 Trading speed for ease *(Direct productivity loss)*

One explanation for developers continuing to use AI assistance throughout the study despite it decreasing their productivity is that they are trading productivity for ease—perhaps the development experience when using AI tools like Cursor is so much cognitively easier that developers choose to use it even though they are slower on average. We qualitatively receive mixed reports from developers regarding how effortful work with AI tools is relative to work without them. One developer notes that completing issues when using AI *"felt like less effort."* Another developer noted that he *"didn't really think about some of the problems"* when using AI. However, another developer noted that there was *"no effort or approach difference [with AI]."*

Quantitatively, we track whether developers continue using Cursor after the experiment period ends, and find that 69% of developers continue using it after the study period has ended. This impressive retention rate suggests that developers are getting *some* significant value from using Cursor, and it seems unlikely this is solely a result of miscalibration on their productivity.

C.2.6 Low quality initial pull requests *(Direct productivity loss)*

One possible explanation for AI-allowed issues taking longer than AI-disallowed issues could be that developers submit lower quality pull requests when using AI (e.g. because they do not fully review or fix lower quality AI-generated code), and then have to spend extra time after PR review making improvements/fixes.

We observe a statistically insignificant difference in the mean post-review implementation time (9 and 15 for AI-disallowed and AI-allowed issues respectively).

However, qualitatively, AI-allowed PRs do not appear to be of lower quality. Because developers have typically spent many years contributing to their respective repositories, they anticipate the high quality PR review standards; 100% of developers report that they "only submit high quality PRs".

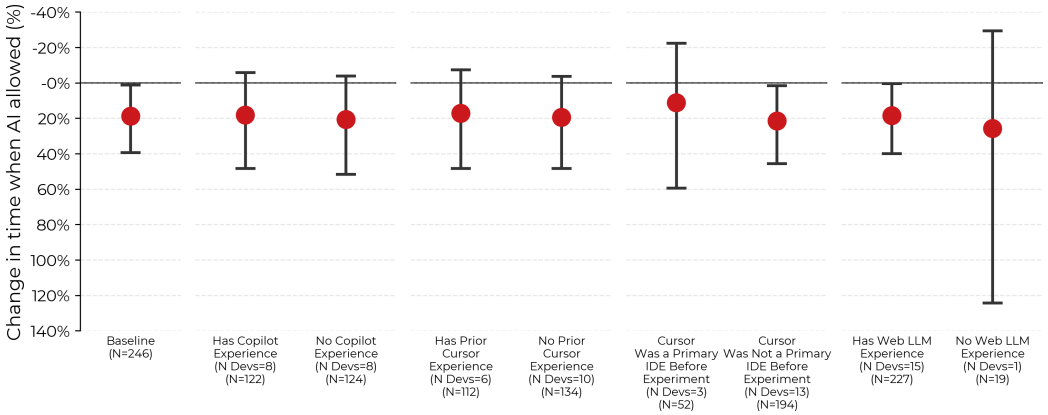


Figure 10: *We evaluate speedup on various subsets of developers' prior experience with GitHub Copilot, Cursor, and web LLMs (e.g. ChatGPT). Developers with prior Cursor experience (who use Cursor in the study) are slowed down similarly to developers without prior Cursor experience, and we see no difference between developers with/without Copilot or web LLM experience. See Section D.3 for details on how we estimate heterogeneous treatment effects.*

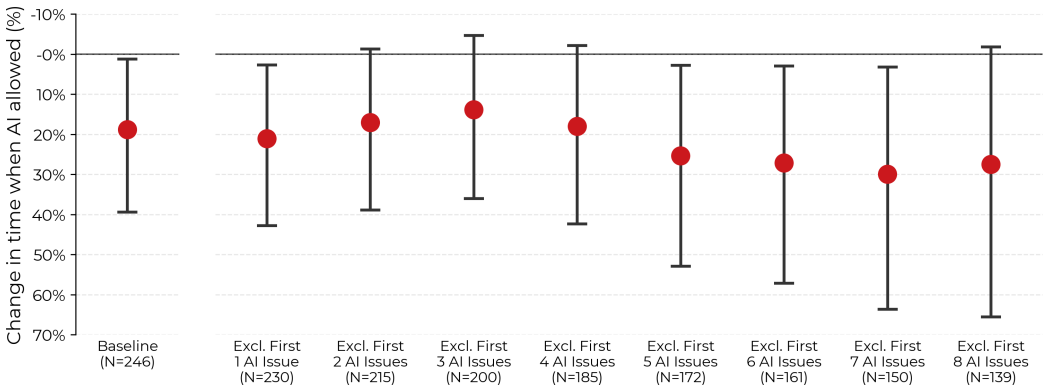


Figure 11: *We see similar slowdown percentages when excluding up to the first eight AI-allowed issues developers work on, suggesting that developers lacking basic skills around using AI effectively does not contribute substantially to the slowdown result. See Section D.3 for details on how we estimate heterogeneous treatment effects.*

C.2.7 Below-average use of AI tools *(Limits AI performance)*

Although all developers have used AI tools previously (most have used LLMs for tens to hundreds of hours), only 44% of developers have prior experience with Cursor. *A priori*, we could imagine significant learning effects for these tools, such that individuals with experience using these tools may be slowed down less than individuals without this experience.

Figure 10 breaks down the percentage change in issue completion time due to AI by different levels of developers' prior experience using AI tools. We don't see meaningful differences between developers based on prior experience with AI tooling.

We further check if developers appear to get better at using AI over the course of the experiment (Figure 11). There does not appear to be a meaningful difference in slowdown when excluding up to the first eight AI-allowed issues each developer completes. This is evidence against the hypothesis that slowdown is caused by our developers lacking basic skills in AI tool use that can be developed in a short period of time.

To more directly assess the impact of learning effects and AI tool use skill on productivity, we estimate speedup on issues bucketed by the number of hours of Cursor experience the developer had

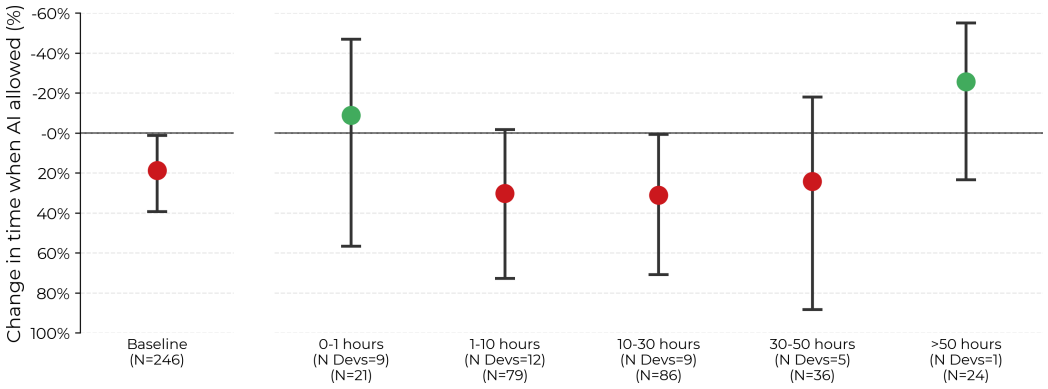


Figure 12: *Speedup on issues where developers have varying hours of experience using Cursor (including prior Cursor experience, plus their usage during the study period). We don't see large differences across the first 50 hours that developers use Cursor, but past 50 hours we observe positive speedup. However, we are underpowered to draw strong conclusions from this analysis. See Section D.3 for details on how we estimate heterogeneous treatment effects.*

when working on the issue (Figure 12). Up to 50 hours of Cursor experience, it broadly does not appear that more experience reduces the slowdown effect. However, we see positive speedup for the one developer who has more than 50 hours of Cursor experience, so it's plausible that there is a high skill ceiling for using Cursor, such that developers with significant experience see positive speedup. As developers spend more time using AI assistance, however, their development skills without AI assistance may atrophy. This could cause the observed speedup to mostly result from weaker AI-disallowed performance, instead of stronger AI-allowed performance (which is the question we're interested in). Overall, it's unclear how to interpret these results, and more research is needed to understand the impact of learning effects with AI tools on developer productivity.

Broadly, we qualitatively observe that developers use Cursor at a level comparable to how well the authors use Cursor for software development, which is largely unsurprising, given we provide training at the beginning of the study, and periodic feedback throughout (Section G.3). While we don't expect that developers are using AI assistance optimally, we do not find evidence that they are below-average in AI tool use ability.

C.2.8 AI generation latency *(Limits AI performance)*

All else equal, faster AI generations would result in developers being slowed down less. Qualitatively, a minority of developers note that they spend significant time waiting on AI to generate code. One developer notes that for *"larger refactorings, [AI generation] takes a couple of minutes"*. Another developer notes that when waiting on AI generations, he *"spends time on Twitter"*. However, not all developers feel majorly affected by this time, for example, one developer notes that he was *"never waiting for more than like 20 seconds."*

Quantitatively, on the subset of 44 issues with valid labeled screen recordings, we find that when AI is allowed, developers spend approximately 4% of their time waiting on AI generated outputs when working with AI. This percentage is small, but non-trivial.

Particularly given the recent benefits seen from inference/test-time compute, there are likely fundamental tradeoffs between AI output latency and performance/reliability. In general, we can imagine a pareto frontier between these variables (either for a given model, or between models/architectures)—but the optimal point on this frontier plausibly depends both on the domain, and on how exactly humans use AI tools to substitute for their labor.

C.2.9 Suboptimal elicitation *(Limits AI performance)*

Cursor is one of the most widely used AI-enabled IDEs. Developers use Cursor agent/composer and Cursor chat in 55% and 48% of AI-allowed issues (respectively), indicating they are reasonably often using Cursor's best (at the time) scaffolding.

However, normal usage of Cursor's AI tools does not typically involve sampling more than a few thousand tokens from models. Recent literature shows that model performance can improve significantly with respect to the number of tokens sampled at inference time [40], so it's natural to wonder if the lack of speedup is driven by limited token spend.

We note that limited token spend would not explain the difference between our results and other work that find positive speedup for programming tasks [10; 17].

However, we can imagine alternative elicitation strategies that effectively use much higher token spend, like sampling many trajectories in parallel from agents and using an LLM judge (or e.g. self-consistency [41]) to filter to the output most likely to be useful for the human. We do not provide evidence about these elicitation strategies, as developers in our study typically use Cursor and web LLMs like chatGPT, so it remains unclear how much effect these strategies would have on developer productivity in the wild.

C.3 Factors unlikely to contribute to slowdown

We provide suggestive evidence that 6 factors are not contributing to observed slowdown, summarized in Table 1.

C.3.1 Unfamiliar development environment *(Experimental artifact)*

Developers qualitatively report VSCode and Cursor to be equivalent development experiences, when AI features in both are turned off[14]. Thus, to make sure they don't use AI features by accident when AI is disallowed, some developers choose to use VSCode for AI-disallowed issues, and Cursor for AI-allowed issues.

However, some developers do use different IDEs entirely for issues where AI is disallowed, and use Cursor for AI-allowed issues. If developers are substantially more productive in these other environments, e.g. because they are more familiar with keyboard shortcuts or have special personalized tooling set-up, then even if AI is helpful, they might be slowed down substantially because they aren't used to using Cursor.

We find evidence that this does not contribute to slowdown. When we restrict to developers who are using comparible IDEs between treatment conditions, we observe a slowdown of 24%.

Furthermore, if Cursor was difficult to use effectively, we might expect to see learning effects as developers personalize it and learn its features. To measure this, we estimate the percentage speedup as developers use Cursor for different lengths of time. Figure 12 does not show meaningful differences in slowdown across the first 50 hours of Cursor usage. Combined with the fact that developers rarely note having any issues or bottlenecks using Cursor, this suggests that Cursor being an unfamiliar environment does not substantially contribute to slowdown.

C.3.2 Cheating or under-use of AI *(Experimental artifact)*

Observing screen recordings provides evidence that slowdown is not driven by developers not using AI when they are allowed to, or because they cheat and do use AI when they are not allowed to.

On the subset of 44 manually labeled valid screen recordings where AI is allowed, we find that developers do not use AI in only 16.4% of them, indicating that AI assistance was typically used on issues where AI was allowed.

We observe 3 specific instances where AI tools were used on AI-disallowed issues, out of 54 labeled AI-disallowed screen recordings, suggesting an overall cheating rate of $\sim 6\%$. At least one instance of observed cheating appears to be accidental.

C.3.3 Issue dropout *(Experimental artifact)*

One potential source of bias are selection effects with respect to which issues are completed by developers. For example, if developers can choose which issues they want to complete after the

[14]This is unsurprising because a) Cursor is a fork of VSCode, and b) Cursor has features to transfer settings, keyboard shortcuts, and other configuration information directly from VSCode to Cursor.

issues are assigned as AI-allowed or AI-disallowed, they may choose to complete only easy AI-disallowed issues (given their expectation that AI will speed them up). This would make AI-allowed issues harder on average, which could contribute to slowdown.

To avoid this, developers commit to completing *all* issues after they are assigned to AI-allowed and AI-disallowed buckets. However, as developers do not work on these repositories as their full-time job, sometimes they fail to meet these commitments. 3 developers fail to complete 7 total issues assigned to AI-disallowed, and 4 developers fail to complete 6 total issues assigned to AI-allowed.

However, this does not appear to be driving results. When we subset developers to those who have no uncompleted AI-disallowed issues, we still find a slowdown of 42%. When we subset to developers who have no uncompleted AI-allowed issues, we find a slowdown of 21%. And when we subset to developers with no dropout on any issues, we find a slowdown of 21%.

Separately, 24 issues were intentionally dropped by developers, the leading reasons being: the issue ended up being too hard to implement (11 issues), the issue was no longer relevant to their work by the time they got to it (5 issues), or the issue was implemented by another developer instead (3 issues). If developers made the choice to drop these issues because they were not allowed to use AI, this selection effect could bias results. However, when asked about de-prioritized issues, developers confirm that the issues are not de-prioritized due to AI usage. We manually review these intentionally dropped issues to confirm that they are not dropped for reasons related to their condition assignment.

C.3.4 Non-robust outcome measure *(Experimental artifact)*

Imputation of Post-Review Time on Unreviewed Issues In Section 2.3, we describe how we impute a small number ($\sim 12\%$) of post-review implementation times for issues that were not reviewed by the completion of the study. These 27 issues are balanced between AI-allowed and AI-disallowed conditions, and the forecasted times to complete are similar, which substantially reduces the likelihood of strong selection effects biasing results.

To quantitatively check the robustness of our post-review time imputation, we re-run our speedup estimation in two ways: 1) we impute AI-allowed post-review implementation times as a constant 1 hour (while AI-disallowed post-review time is imputed as 0), and 2) we impute AI-disallowed post-review implementation times as one hour (while AI-allowed is set to 0). This is intended to simulate the case if post-review times for unreviewed issues were systematically biased for one condition or the other, and because the mean post-review implementation times are 15 and 9 minutes for AI-allowed and AI-disallowed issues respectively, this would represent a very extreme bias. When imputing using this extreme bias, we still do not observe substantial impacts on our speedup estimates in either case (14% and 23% slowdowns when using AI, respectively).

Self-Reported Times To validate that there is not significant bias in self-reported times that contributes to slowdown, we also measure the slowdown percentage computed using the total length of screen recordings associated with a given issue, instead of using self-reported times. We observe a 25% slowdown when using total screen recording time instead of the self-reported time spent, which is similar to the 24% we observe when estimating speedup from self-reported implementation times on the same subset of issues.

C.3.5 Non-robust estimator *(Experimental artifact)*

We evaluate the slowdown implied by alternative estimators, in addition to the regression-based estimator that yields our primary results.

In Figure 13 we report the slowdown observed when using four alternative estimators other than our baseline regression described in Appendix D. We compare a simple ratio estimator, which reports $\frac{\text{Mean(time with AI allowed)}}{\text{Mean(time with AI disallowed)}} - 1$ (Section D.4, as well as three other regression-based estimators with different covariate specifications (described in Figure 13's caption).

The alternative estimators all report similar results, suggesting that slowdown is robust to our particular estimator specification.

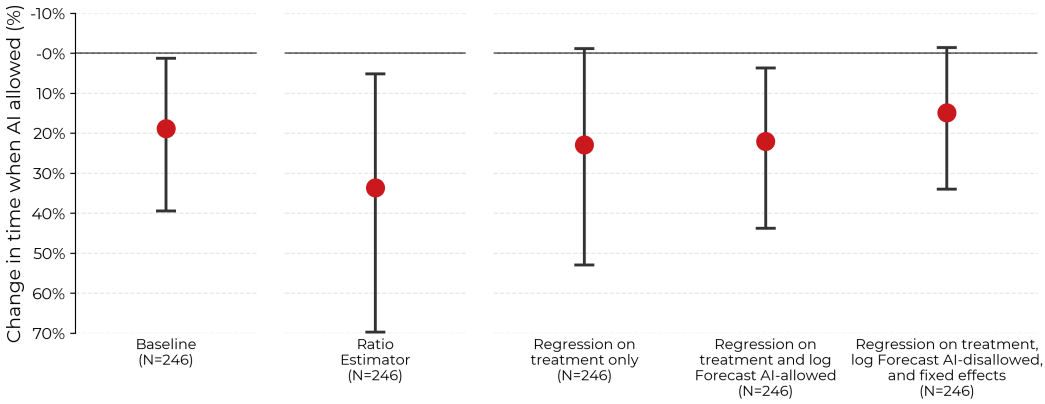


Figure 13: *In addition to a basic ratio estimator, we also evaluate speedup when regressing* log*(implementation time) on whether AI is allowed; whether AI is allowed and the log forecast of implementation time with AI-allowed; and whether AI is allowed, the log forecast of implementation time with AI-disallowed, and developer fixed effects.*

C.3.6 Non-frontier model usage *(Limits AI performance)*

Given the significant rate of progress in AI (particularly in software/coding domains), one explanation for slowdown could be that developers used older, less capable models on AI-allowed issues.

This does not appear to be the case. Across the 44 valid labeled AI-allowed loom videos, we find that developers used Claude 3.7 Sonnet (thinking mode), Claude 3.7 Sonnet, and Claude 3.5 Sonnet in 25%, 34%, and 23% of issues respectively. Other models used are GPT-4o (11%), Gemini 2.5 Pro (3%), and o1 (2%). Developers prompt no AI model 16.4% of the time. These percentages do not include AI autocomplete, which developers use on the vast majority of AI-allowed issues.

Most issues were completed in February and March 2025, before models like Claude 4 Opus or Gemini 2.5 Pro were released.

D Empirical Strategy

D.1 Regression

For each issue i we observe the realised completion time $T_i > 0$, a binary treatment flag $\mathrm{AI}_i \in \{0, 1\}$, and the developer's *ex-ante* forecast of how long the task would take without AI, denoted $\widehat{T}_i^{\mathrm{NoAI}} > 0$.

We estimate the log-linear model

$$\log T_i = \alpha + \beta\, \mathrm{AI}_i + \delta\, \log \widehat{T}_i^{\mathrm{NoAI}} + \varepsilon_i, \tag{1}$$

via ordinary least squares and report homoskedastic standard errors[15]. Random assignment of AI_i guarantees consistency of $\hat{\beta}$ for $\beta = \mathbb{E}[\log T \mid \mathrm{AI} = 1] - \mathbb{E}[\log T \mid \mathrm{AI} = 0]$.

We include forecasts as a control variable because they serve as a proxy for issue difficulty and are highly predictive of completion times. This substantially increases our statistical power without introducing bias, as forecasts were elicited prior to treatment assignment and thus cannot be affected by treatment status.[16]

Figure 14 displays regression diagnostics associated with this specification.

[15] We do not find heteroskedasticity in our data and, empirically, heteroskedastic standard errors have almost identical width to the homoskedastic errors we report.

[16] We do not generally include developer fixed effects because they explain minimal variation in the outcome conditional on forecasts. Section C.3.5 displays estimates from a regression specification including developer fixed effects.

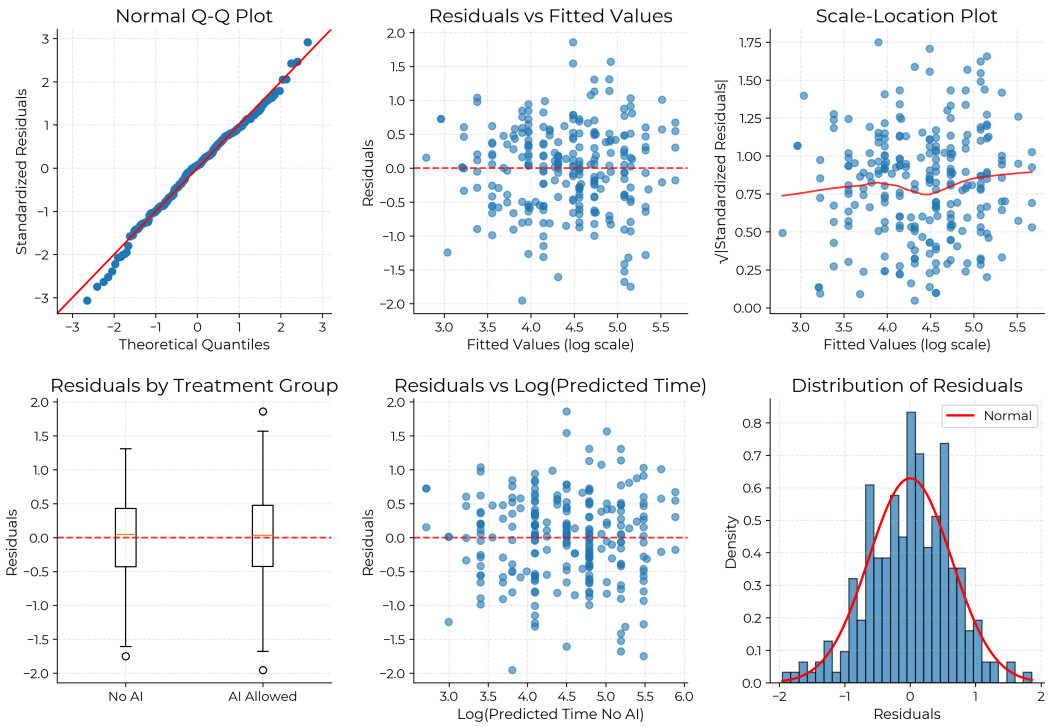


Figure 14: *Regression diagnostics associated with the regression specification in equation* (1).

D.2 From log ratios to speedup

Taking conditional expectations of equation (1) given treatment status and forecast:

$$\mathbb{E}[\log T \mid \text{AI} = 1, \widehat{T}^{\text{NoAI}}] = \alpha + \beta + \delta \log \widehat{T}^{\text{NoAI}} \tag{2}$$

$$\mathbb{E}[\log T \mid \text{AI} = 0, \widehat{T}^{\text{NoAI}}] = \alpha + \delta \log \widehat{T}^{\text{NoAI}} \tag{3}$$

Therefore, $\beta = \mathbb{E}[\log T \mid \text{AI} = 1, \widehat{T}^{\text{NoAI}}] - \mathbb{E}[\log T \mid \text{AI} = 0, \widehat{T}^{\text{NoAI}}]$ represents the average treatment effect on the log scale, conditional on forecast.

To convert to a speedup measure, we note that:

$$\exp(\beta) = \frac{\mathbb{E}[T \mid \text{AI} = 1, \widehat{T}^{\text{NoAI}}]}{\mathbb{E}[T \mid \text{AI} = 0, \widehat{T}^{\text{NoAI}}]} \tag{4}$$

$$\text{S} = \exp(\beta) - 1 \tag{5}$$

(Step (4) holds provided the disturbance ε_i is independent of both AI_i and $\widehat{T}_i^{\text{NoAI}}$ and that $\mathbb{E}[e^{\varepsilon_i}]$ exists.[17])

Then under standard regularity assumptions (which our diagnostics suggest hold) our OLS estimator $\hat{\beta}$ is normally distributed $\mathcal{N}(\beta, \sigma^2)$ for some σ which we can estimate using it's standard error $\text{SE}[\hat{\beta}]$. This lets us construct a 95% confidence interval for β in the usual way:

$$\text{CI}_{95\%} = \left[\hat{\beta} - 1.96 \cdot \text{SE}[\hat{\beta}], \hat{\beta} + 1.96 \cdot \text{SE}[\hat{\beta}]\right] \tag{6}$$

[17] Because $T_i = e^{\alpha} e^{\beta\, \text{AI}_i} (\widehat{T}_i^{\text{NoAI}})^{\delta} e^{\varepsilon_i}$, taking expectations conditional on AI_i and $\widehat{T}_i^{\text{NoAI}}$ gives $\mathbb{E}[T \mid \text{AI} = j, \widehat{T}^{\text{NoAI}}] = e^{(\alpha+\beta j)} (\widehat{T}^{\text{NoAI}})^{\delta}\, \mathbb{E}[e^{\varepsilon_i}]$. The common factor $\mathbb{E}[e^{\varepsilon_i}]$ cancels when we form the ratio of these conditional means, yielding $\mathbb{E}[T \mid \text{AI} = 1, \widehat{T}^{\text{NoAI}}] / \mathbb{E}[T \mid \text{AI} = 0, \widehat{T}^{\text{NoAI}}] = \exp(\beta)$.

As $\mathrm{S} = \exp(\beta) - 1$ is a monotonic function of β we can construct a confidence interval for S by simply applying the function to the endpoints of β's confidence interval:

$$\mathrm{CI}_{95\%} = \left[e^{\hat{\beta} - 1.96 \cdot \mathrm{SE}[\hat{\beta}]} - 1, e^{\hat{\beta} + 1.96 \cdot \mathrm{SE}[\hat{\beta}]} - 1\right] \tag{7}$$

D.3 Heterogeneous treatment effects

To test for differential treatment effects across subgroups, we estimate models with interaction terms. For a binary characteristic X_i (e.g., prior Cursor experience), we estimate:

$$\log T_i = \alpha + \beta_1 \,\mathrm{AI}_i + \beta_2 \, X_i + \beta_3 \,(\mathrm{AI}_i \times X_i) + \delta \, \log \widehat{T}_i^{\mathrm{NoAI}} + \varepsilon_i \tag{8}$$

Taking conditional expectations:

$$\mathbb{E}[\log T \mid \mathrm{AI} = 1, X = 0, \widehat{T}^{\mathrm{NoAI}}] - \mathbb{E}[\log T \mid \mathrm{AI} = 0, X = 0, \widehat{T}^{\mathrm{NoAI}}] = \beta_1 \tag{9}$$

$$\mathbb{E}[\log T \mid \mathrm{AI} = 1, X = 1, \widehat{T}^{\mathrm{NoAI}}] - \mathbb{E}[\log T \mid \mathrm{AI} = 0, X = 1, \widehat{T}^{\mathrm{NoAI}}] = \beta_1 + \beta_3 \tag{10}$$

Thus, the treatment effect for the $X = 0$ group is β_1, while for the $X = 1$ group it is $\beta_1 + \beta_3$. We similarly transform these to speedup measures: $\mathrm{S}_{X=0} = \exp(\beta_1) - 1$ and $\mathrm{S}_{X=1} = \exp(\beta_1 + \beta_3) - 1$.

To construct confidence intervals for these subgroup effects, we test linear hypotheses of the form $L^T \boldsymbol{\theta} = c$. For the $X = 1$ group effect, $L^T = [0, 1, 0, 1, 0]$ selects $\beta_1 + \beta_3$. Using homoskedastic standard errors, the Wald statistic:

$$W = \frac{L^T \hat{\boldsymbol{\theta}} - c}{\sqrt{L^T \hat{V} L}} \sim \mathcal{N}(0, 1) \tag{11}$$

where $\hat{V}$ is the variance-covariance matrix. This asymptotic normality allows us to construct confidence intervals using the same critical values. For the linear combination $\hat{\beta}_1 + \hat{\beta}_3$:

$$\mathrm{CI}_{95\%} = \left[(\hat{\beta}_1 + \hat{\beta}_3) - 1.96 \cdot \mathrm{SE}[\hat{\beta}_1 + \hat{\beta}_3], (\hat{\beta}_1 + \hat{\beta}_3) + 1.96 \cdot \mathrm{SE}[\hat{\beta}_1 + \hat{\beta}_3]\right] \tag{12}$$

We then apply the same transformation as above to obtain confidence intervals for S:

$$\mathrm{CI}_{95\%} = \left[e^{(\hat{\beta}_1 + \hat{\beta}_3) - 1.96 \cdot \mathrm{SE}[\hat{\beta}_1 + \hat{\beta}_3]} - 1, e^{(\hat{\beta}_1 + \hat{\beta}_3) + 1.96 \cdot \mathrm{SE}[\hat{\beta}_1 + \hat{\beta}_3]} - 1\right] \tag{13}$$

D.4 Ratio estimator

For robustness, we also compute a simple ratio-of-means estimator: $\hat{S}_{\text{ratio}} = \frac{\bar{T}_{\text{AI-allowed}}}{\bar{T}_{\text{AI-disallowed}}} - 1$, where $\bar{T}$ denotes the sample mean of implementation times. We obtain 95% confidence intervals using quantiles of $\hat{S}_{\text{ratio}}$ from a pooled bootstrap.

We use this same estimator to calculate aggregate percentage change in implementation times implied by developer forecasts. Note that this procedure is distinct from averaging over percentage changes in implementation time per-task.

D.5 Participant post-study estimator

After the study period, developers estimate post hoc their speedup from AI across their completed issues. To make these estimates comparable to their forecasts (which they provide before each issue is randomized to allow or disallow AI), we compute a weighted arithmetic mean of the ratios of times they report. Each developer's ratio r_i is weighted proportionally to the number of issues n_i they completed:

$$\bar{r}_w = \frac{\sum_{i=1}^{N} n_i r_i}{\sum_{i=1}^{N} n_i} \tag{14}$$

We calculate the standard error of this weighted mean using the weighted variance formula:

$$\text{SE}(\bar{r}_w) = \sqrt{\frac{\sum_{i=1}^{N} w_i (r_i - \bar{r}_w)^2}{\left(\sum_{i=1}^{N} w_i\right)^2}} \tag{15}$$

where $w_i = n_i$ are the issue-based weights. The 95% confidence interval around the weighted mean is then derived using the standard normal approximation: $\bar{r}_w \pm 1.96 \times \text{SE}(\bar{r}_w)$.

E Other Analysis

E.1 Balance and proportions checks

Characteristic	AI-allowed (n=136)	AI-disallowed (n=110)	Difference	p-value
AI-disallowed forecasted time	116 (80)	106 (63)	9	0.28
AI-allowed forecasted time	87 (62)	82 (53)	4	0.56

Table 4: *Balance table comparing forecasted issue completion times between AI-allowed and AI-disallowed groups. Means are shown with standard deviations in parentheses. P-values are from two-sided Welch t-tests. There is no meaningful difference in forecasted difficulty, indicating successful randomization.*

Treatment Group	Intended Proportion	Actual Proportion	Sample Size
AI-allowed	50%	55.3% (3.2%)	136
AI-disallowed	50%	44.7% (3.2%)	110
P-value		0.10	

Table 5: *Treatment assignment proportions. Actual proportions are shown with standard errors in parentheses. P-value is from chi-square test for deviation from intended 50/50 allocation.*

Table 4 confirms that randomization leads to balanced average difficulty between AI-allowed and AI-disallowed groups. Table 5 shows that the difference in the proportion of issues randomly assigned to AI-allowed and AI-disallowed groups is not statistically significant at common p-value thresholds.

E.2 Per-developer speedup and forecast calibration

Figure 15 shows the relationship between the times developers forecast issues will take without AI (which we interpret as a forecast of issue difficulty), and how long the issues actually end up taking them (colored by whether AI was allowed or disallowed for each issue). We can see that the median forecasted implementation time is almost identical across treatment conditions, while AI-allowed issues take longer on average than AI-disallowed issues (note the log axes).

We estimate speedup per developer (Figure 16) using our standard methodology for estimating heterogeneous effects (Section D.3). 75% of developers experience slowdown.

Interestingly, despite developers reliably forecasting incorrectly that AI-allowed issues will take less time, they are still calibrated in a relative way on the speedup from AI (Figure 17). Specifically, on issues that developers predict significant speedup (i.e. $\geq$ 33% or the top tertile of forecasted speedup), developers are not slowed down by AI, and slowdown monotonically decreases as the forecasted speedup increases.

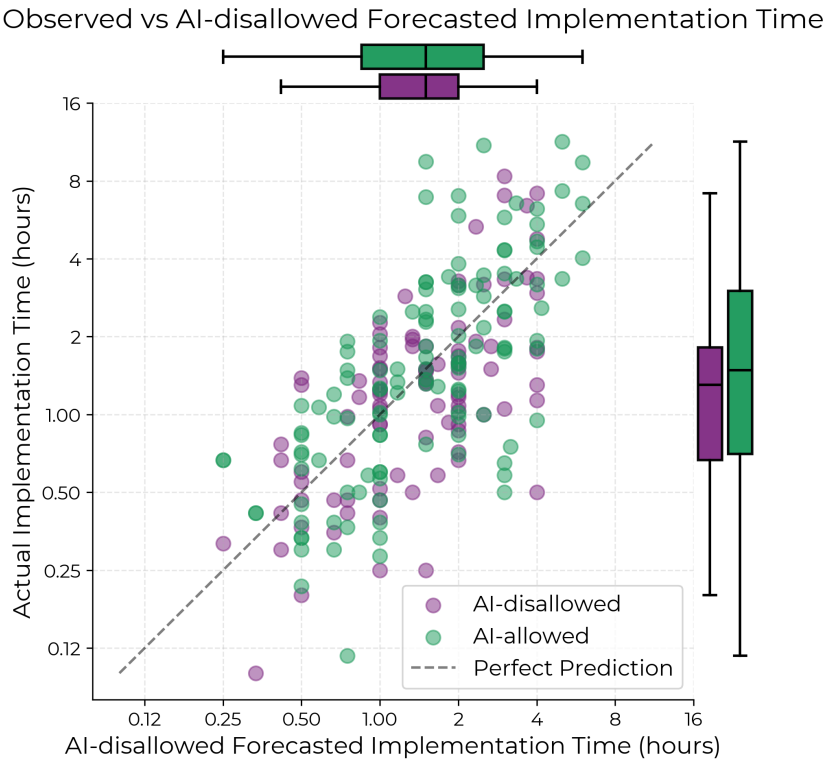


Figure 15: *Distributions of issue implementation time with AI allowed and disallowed as a function of the forecasted implementation time without AI.*

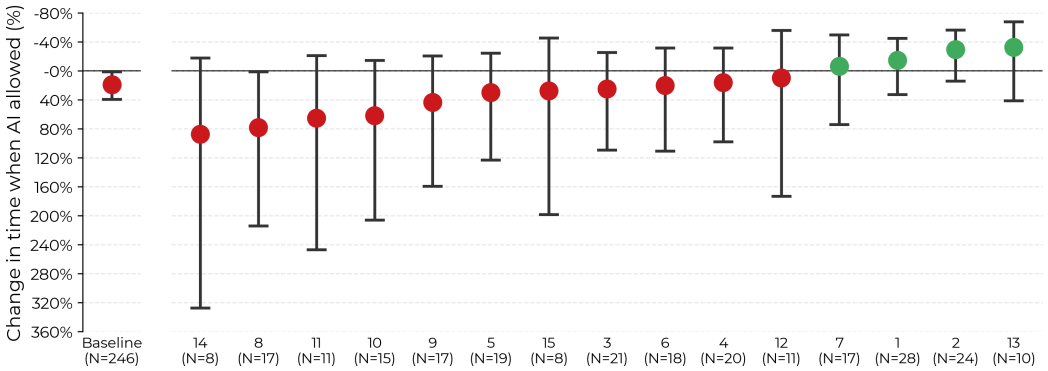


Figure 16: *Speedup estimates per developer. Developer IDs correspond to the developer's rank by number of issues in our data.*

E.3 Randomization

There were 25 issues early in the study that were randomized using a block randomization scheme intended to increase statistical power. Developer issue lists ended up being too small for this strategy to be viable, so we abandoned it early on in favor of simply using a simulated fair coin flip.

Excluding these issues does not affect our result—we still find a slowdown of 20%. Given this, we include these issues in our analysis and results to increase statistical power.

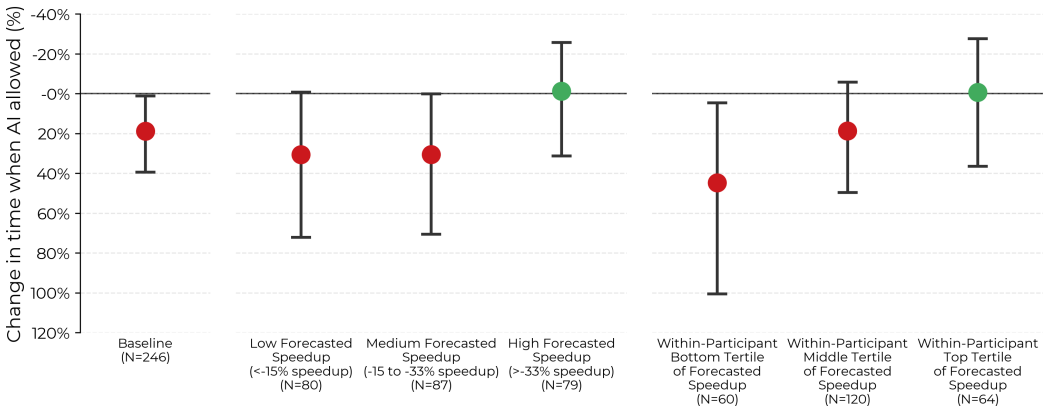


Figure 17: *Speedup broken down by forecasted speedup between and within developers (developers forecast how long they expect each issue to take with and without AI). Speedup cutoffs are chosen to make bins approximately similarly sized. Tertiles are imbalanced because forecasted speedup contains duplicates that we assign to a single bin. Developers experience less slowdown on issues that they forecast high speedup. See Section D.3 for details on how we estimate heterogeneous treatment effects.*

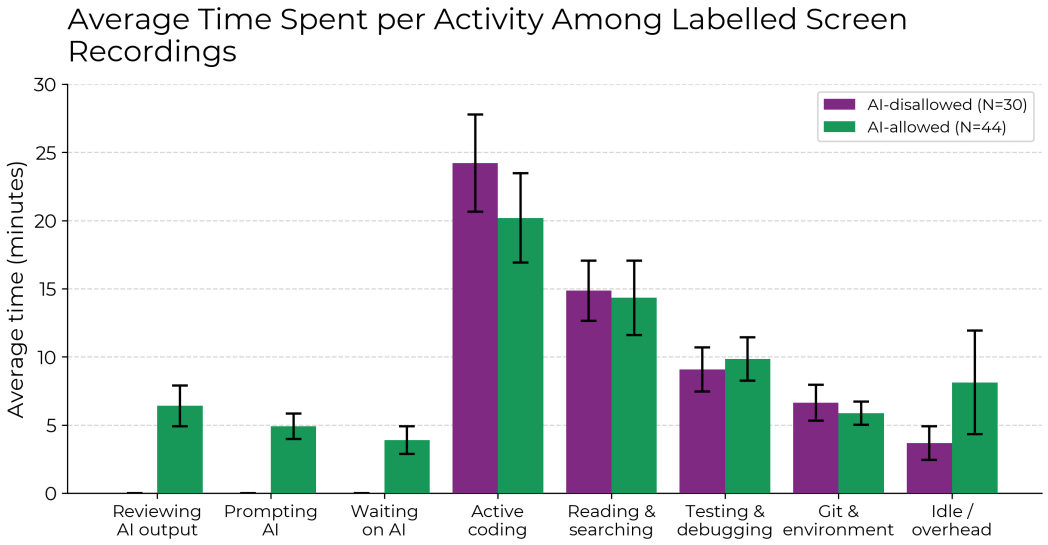


Figure 18: *Average minutes spent on different high-level activities when completing issues with AI-allowed vs. AI-disallowed.*

E.4 Fine-Grained Screen Recording Labels

Figures 18, 19, and 20 present various breakdowns of the time developers spend on different activities as they work. See Section 2.4 and Section G.8 for more detail on screen recording labels.

E.5 Expert forecasts

We display summary statistics regarding expert forecasts of our result. Recall that we originally elicited expert forecasts for $\frac{\mathbb{E}[T \mid \text{AI=0}]}{\mathbb{E}[T \mid \text{AI=1}]}$, but here report results on the $\frac{\mathbb{E}[T \mid \text{AI=1}]}{\mathbb{E}[T \mid \text{AI=0}]} - 1$ scale we use for Figure 1.

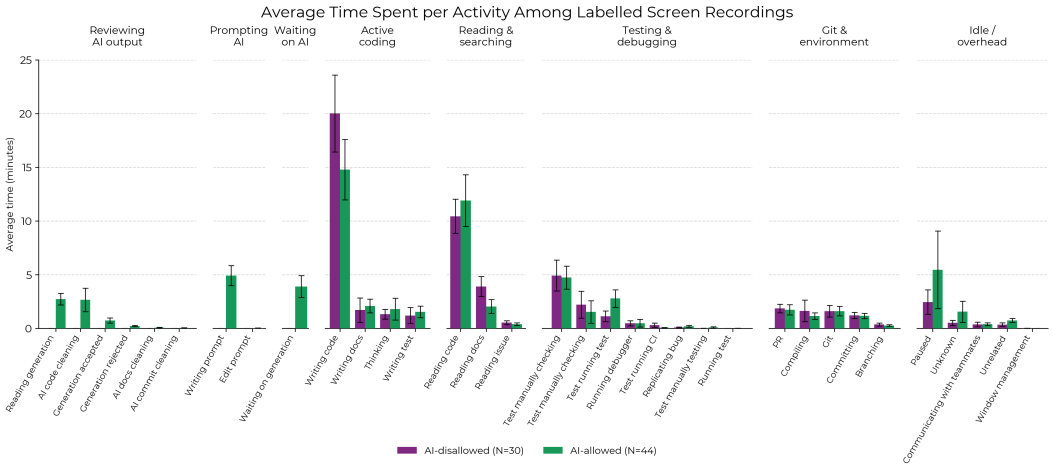


Figure 19: *Average minutes spent across 27 fine-grained activity categories.*

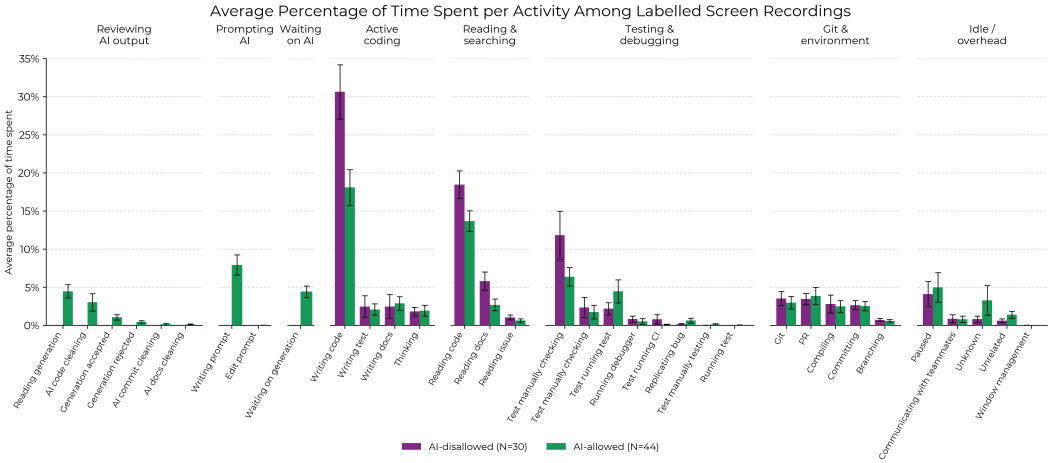


Figure 20: *Percentage of time spent on fine-grained activities when AI is allowed vs. disallowed.*

E.6 Other treatment effects

Figure 21 displays estimates using alternative outcome measures or subsets of our data. Figure 22 displays treatment effects by the calendar month in which an issue implementation was started.

F Open-Source Development and AI Tooling Primers

F.1 Open-Source Development

An open source software (OSS) project is typically defined by a *repository*, which is a collection of code and assets. The repository for the popular pandas Python package, for example, can be found here.

Expert Group	N	Mean	Min	P25	P50	P75	Max
Economics	34	-38.7	-80.0	-56.0	-37.5	-26.3	81.8
Machine Learning	54	-38.0	-88.9	-55.5	-33.3	-20.2	0.0

Table 6: *Expert Forecast Statistics*

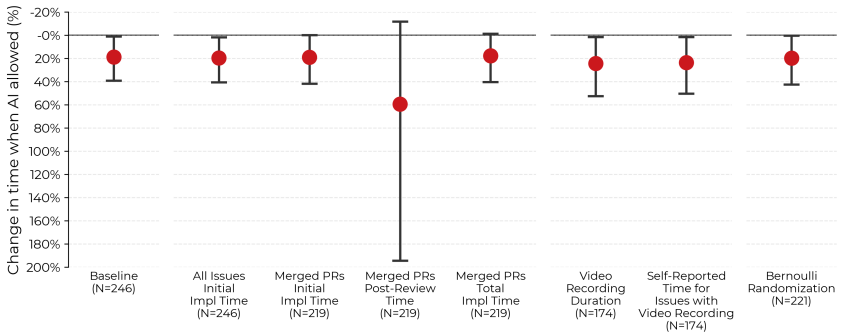


Figure 21: *Speedup by alternative outcomes measures or subsets of our data. The Bernoulli randomization subset excludes the 25 issues randomized using a block randomization scheme (see Section E.3).*

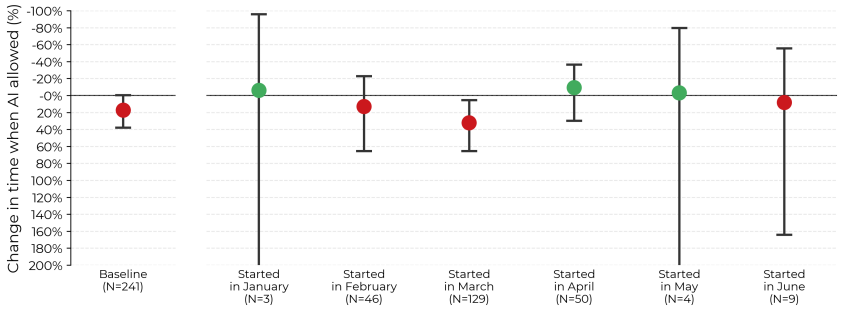


Figure 22: *Speedup by month issue implementation started, as measured by first commit. Confidence intervals for January and May effects are cut-off for readability; the lower bounds are at approximately 1800%. See Section D.3 for details on how we estimate heterogeneous treatment effects.*

Any developer who contributes to a given repository is known as a *contributor*. Active contributors for the pandas library are listed here.

Contributors to an OSS project work off of *issues*, which implicitly or explicitly describe tasks (bugs to fix, features to build, etc.) tracked within the repository. A few example pandas issues include `BUG: to_dict(orient='dict') does not convert np.nan to None in Pandas 2.2.3` (link) and `ENH: Enable nsmallest/nlargest on object dtype.` (link).

Contributors resolve an issue by submitting a *pull request* (PR), which is a proposal to make changes to the repository. For example, this PR to the pandas repository fixes a bug by `Clip[ing] corr edge cases between -1.0 and 1.0`, addressing the corresponding issue `BUG: .corr() values significantly higher than 1.` (link).

After a contributor opens a PR, another contributor (often a maintainer) will review the PR. *PR review* consists of reading and testing the code changes while paying attention to correctness, performance, and repository-specific code style. The reviewer may leave comments requesting changes; in the above PR, the author was asked if other functions needed a similar bug fix (link).

After a PR is reviewed, the original contributor may make changes to address review comments. Multiple rounds of review may occur, although this is rare. After all review comments are addressed, the PR is merged into the repository. This results in the issue being marked as completed or closed.

Though this description of open-source software development is a reasonable default, diversity abounds. Some projects have many contributors, others only have a single contribute; some contributors do in-depth reviews, others merge in PRs without review at all.

Stars indicate the number of developers who have expressed interest in the repo. *Forks* indicate the number of copies of the repository that have been made by developers so they can make their own modifications. Stars and forks can be seen as measures of repo popularity. The pandas library has 45,200 stars and 18,400 forks, making it an extremely popular repository.

F.2 Primer on AI Tooling

F.2.1 Web Interfaces

Many companies training large language models (LLMs) offer web-based user interfaces wherein users can chat with AIs. For example, users can interact with OpenAI models at chatgpt.com, Google DeepMind models at gemini.google.com, and Anthropic models at claude.ai. During our study period, popular LLMs offered by these developers include OpenAI's GPT-4o, GPT-4.5, o1, o3-mini, o3, and o4-mini, Google DeepMind's Gemini 2.5 Flash and Gemini 2.5 Pro, and Anthropic's Claude 3.5 Sonnet (New) and Claude 3.7 Sonnet (although many/most issues were completed before the more recent models were released).

F.2.2 Cursor

Cursor is an integrated development environment (IDE) or 'code editor'—a desktop application from which developers write and otherwise interact with code. It is a fork of the most popular code editor, Visual Studio Code (VSCode) [42].

Cursor being a fork of VSCode enables developers to transfer their workflows from VSCode to Cursor to retain existing extensions and settings that they are most familiar with. Low switching costs are deepened by strong similarities in user interface and features between the two IDEs.

Relative to VSCode, Cursor is notable for having well-integrated AI tools, in particular "Cursor Chat" (previously separated into "agent mode" and "chat mode") and performant AI-powered autocomplete features.

F.2.2.1 Chat and Agent Mode

Cursor Chat allows users to prompt LLMs to make changes from inside the IDE. This LLM has tools enabled allowing it to autonomously explore your codebase, read documentation, run commands, and edit files. In practice, this LLM will generate code attempting to satisfy your prompt, and then show an in-file highlighted view of code changes that users can choose to accept or reject.

(Previously "agent mode" was very similar to Cursor Chat, and "chat mode" was more similar to LLM web-based user interfaces, except model-agnostic and existing inside the IDE.)

The AIs that users typically interact with in Cursor are functionally identical to those they might interact with via web-based user interfaces, except for their additional access to relevant information in the repository, and often additional tool use that allows them autonomously run, test, and debug code they (or others) have written.

F.2.2.2 AI Autocomplete

Traditional IDEs have autocomplete functionality that suggests code completions as you type, primarily by fuzzy-matching on existing defined names in your codebase. For example, if you have a defined function called `add_two_numbers` and then later begin typing `add_tw`, traditional autocomplete will suggest that you finish the completion with `o_numbers`.

AI autocomplete is a feature in both VSCode and Cursor that uses an LLM to suggest edits to code as you write, and goes well-beyond just suggesting previously defined names. For example, if you started by defining a function with the signature `add_two_numbers(a, b):`, AI autocomplete would suggest a completion like `return a + b`.

G Recruitment and Onboarding

Open-source developers are recruited through a multi-stage process to select for active contributors to repositories that had more than 500 stars. Initial outreach was conducted via professional networks, ML-focused communities (Reddit's r/Python, r/MachineLearning), and through GitHub profiles.

- GitHub profiles are found by searching GitHub for the 250 most popular repositories, as well as those tagged with: ai, llm, deep-learning, neural-networks.
- Contributors to these repositories are filtered to exclude those who had committed fewer than than five times in the previous three months.

51 developers filled out a preliminary interest survey, and we further filter down to about 20 developers who had significant previous contribution experience to their repository and who are able to participate in the study. Several developers drop out early for reasons unrelated to the study.

These developers are then given access to Cursor Pro. We conduct a live 30-minute call with each developer where we provide a data collection template, answer basic questions about the experiment and their instructions, and give them training on how to use Cursor. Developers are considered trained once they can use Cursor agent mode to prompt, accept, and revert changes to a file on their own repository.

Additionally, for the duration of the study, we periodically provide feedback to developers on their implementation notes and video recordings. We occasionally email developers with tips on how to use Cursor more effectively if we notice low-hanging fruit (e.g. reminding developers to explicitly tag relevant files when prompting agents) from reviewing their screen recordings.

G.1 Incentivization Scheme

We pay developers $150 per hour to participate in the study. Developers spend the majority of this time implementing issues, with fewer than five hours going to study overhead, including the onboarding call, check-in/feedback calls, the exit interview/survey, and the time they spend collecting their lists of issues.

An alterative incentivization scheme could give developers bonuses for completed issues, to incentivize developers to work as quickly as possible. However, this could cause developers to break issues into smaller chunks (e.g. to increase the total number of issues they complete) or reduce their quality standards to finish work more quickly, which could bias results. We expect that paying developers per hour overall has minimal effect on their behavior, beyond encouraging them to participate in the study.

G.2 Developer Instructions and Survey Data

G.2.1 Developer Instructions

Overview

METR is seeking software engineers who regularly work on large open-source projects to test the effectiveness of AI software engineering tools.

Apply here (bit.ly/ai-speedup-apply)

Eligibility:

You must:

1. Have at least 1 year of professional experience as a software engineer
2. Have at least 6 months experience as an active maintainer of the repository
3. The repository you work on must be:

(a) Open source
(b) At least 500 stars on GitHub or be manually reviewed by METR staff and deemed a high-quality, mature codebase (we know many good code bases don't have a lot of stars)
(c) Have at least 3,000 lines of code (written by humans/in a major programming language, data etc doesn't count)
(d) Have some kind of list of projects to improve it which would take between a few minutes to a few days, and which are relatively independent (i.e. a list of issues to fix, a list of features you intend to add, a general kanban board, etc). It's ok if you make this list specifically for this experiment.

4. Nice-to-haves:
   (a) The codebase is relevant to AI research and development or AI capabilities

Compensation:

1. The total time commitment from a participant is a minimum of 20 hours, but we are interested in larger commitments.
2. We will pay you $150 per hour. Note that during this experiment you will be working on tasks you'd already want to work on in your open source repository. We will be slightly randomizing the order of these tasks as well as what AI tooling you can use [note: we didn't do this, and we clarified this with developers before they began their work], but we don't expect this to be a large impediment to your work.
3. This pilot study will last between 1-2 months, and we will limit funded development hours around 40.

Wait, how does it work?

1. Engineers will start by selecting a set of issues/to-dos from their open source repositories that they are looking to solve.
2. METR will then randomize these tasks into two buckets - on one set of issues, AI is allowed, and on the other set of issues, AI won't be allowed.
3. You'll work through these issues in whatever order you want - just making sure to only use AI when it's allowed.
4. As a participant, you will be doing work of your choosing on a repository of your choosing. This experiment will only change the order of tasks that you do and what LLMs you can use (including potentially restricting you to no LLMs)
5. We are very flexible on when you complete these tasks. You can choose the date and time that works best for you (including weekends!).
6. See more details in the Detailed Timeline.

Why this work matters:

1. AI models are becoming increasingly capable and automating parts of the workforce. We want to understand if or when it could reshape software engineering so we can predict and prepare for its effects.
   (a) In particular, we want to know when models might greatly speed up AI R&D work, creating a feedback loop that would greatly accelerate AI progress
2. Models are traditionally evaluated using simple, artificial benchmarks, where they are tested on their ability to answer multiple choice questions or fix some basic test cases in a Python library. These benchmarks:
   (a) Fail to measure how much models actually speed up engineers in their real workflow, the main real-world use-case for AI right now - and this is exactly what we're attempting to measure with this experiment.

(b) Are typically artificial or have many tasks with no right answer, and lack the nuances and detail of real-world software engineering work
(c) Often require building "scaffolding" for the agents to autonomously write code etc. This scaffolding can be hard to develop and often means the AIs get stuck in places because of silly scaffolding issues. If a human is using the LLM, the scaffolding matters less (and is already widely commercially available) and the human can help get the LLM unstuck
(d) Get saturated quickly because the space between "a model can make any progress on a task at all" and "the model can do almost perfectly at the task" is small. Having the model speed up humans might address this issue because even very weak models can provide some human speedup and even very powerful ones are currently a ways away from being able to replace humans entirely.

3. Although it isn't a primary motivation of our work, we expect you might personally find it useful to know if AI actually speeds you up!

Detailed timeline

Welcome to the METR Human Uplift Pilot!

This document contains an in-order list of the steps in this Uplift Study. Please feel free to leave any comments on the document.

The Steps

Step 1: We have an introduction call

We'll have an introduction call, where I can give you an overview of the experiment and answer any questions you may have. You can book an introduction call here.

If you're interested in moving forward, we'll schedule a kickoff call (see below) for later that week

Step 2: You collect issues

Once you're onboard, it's time to make an issues list. The issue list can come in any format: an email, a Google Sheet, a Github project board. Make it whatever format is easiest for you. Each issue should contain:

1. A description. I don't need to understand this, so feel free to keep it short.
2. A label: bug fix, new feature, exploration, or refactor.
3. Two time estimates:
   (a) No AI Time estimate:if you didn't use any AI tools, how long would this take you?
   (b) AI Time Estimate: if you did use AI tooling to the best of your ability, how long would this take you?
4. Task Expertise:
   (a) Prior Task Exposure: Rate your previous experience with this specific type of task.
      i. 1: Never done this type of task before
      ii. 2: Have seen this type of task done but never done it myself
      iii. 3: Have attempted this type of task once before
      iv. 4: Have done this type of task multiple times before
      v. 5: I am an expert at this type of task
   (b) External Resource Needs: how much documentation/reference material/research will you need to complete this task?
      i. 1: I need extensive documentation / reference material / googling to complete this task.

ii. 2: I would need an occasional documentation / reference check / googling to complete this task.
iii. 3: I could complete this task entirely from memory and experience.

5. Ideally, these issues should be less than 4 hours. If you can break them into $\leq$2 hour tasks, this would be ideal. If larger issues can reasonably be broken down into smaller PRs, feel free to take big issues and break them down into relevant steps.
6. You should have at least 10 issues, and aim for at least 20 hours of issues, and up to 40 hours.

Step 3: You send me issues, I bucket them

Once your issue list is done, you can send it to me. I'll randomize this issue list into two buckets:

1. AI bucket: you can use AI to help you on these issues
2. No AI Bucket: you cannot use AI on these issues.

Step 4: We have a kickoff meeting During the kickoff meeting, I'll give you:

1. The bucketed issue list
2. Access to Cursor Pro (if you don't already have it) as well as a basic training
3. Access to Loom so you can record your screen.
4. The Code of Conduct and Consent form
5. Additionally, I can answer any final questions you might have about this experiment.

Step 5: You work on issues

You're ready to start now. This should mostly look exactly like your normal work.

1. You can work on the issues in any order you like.
2. You can work using any tools you like.
3. However, if an issue is labeled "No AI", then don't use any AI tooling.

You'll record the data described here as you implement these issues.

Note: we will not share Loom videos without any humans outside of METR. We may watch them for quality control or use private LLMs to analyze these videos.

Step 6: Checkin Call We'll have one quick check in call to see how you're doing, resolve any issues, and make sure we're making progress.

Step 7: Get Paid At the end of your issues, you'll get paid. You'll get $150/hour for the number of hours you worked on tasks and created your issues - with a limit of 2 hours for issue creation.

Data to Collect As you implement the issues in this project with and without AI, here is the additional information that you should collect.

Implementation Notes The most important implementation note: if you're working on an issue where AI is allowed, please record which models you use, and where you use them.

Other information is really useful to record as well. Please record any useful notes about the implementation that might be interesting for this study. For example:

1. "Cursor implemented most of this code, with just a simple prompt from me."
2. "Cursor edited my package.json and I didn't notice, which caused me to lose 30 minutes fixing dependencies."

3. "Not being able to use AI was tough, as there was a lot of boilerplate could I could have easily auto-generated"

Link to PR Link to the final PR that you implemented to solve this issue.

Note that if you implement a fix to multiple issues within one PR, just make sure to tag which commits correspond to which changes in that one PR.

Screen Recording Link Link to a screen recording of the implementation of this issue. We ask that participants record their screen for all of the issues that they work on.

Time Tracking We ask that you track two separate time categories: initial implementation time and post-review implementation time. These two time categories should sum to the "total amount of time it took for you to complete this feature to the point that it was mergable into the codebase."

Initial Implementation Time How long did it take you to get the PR up for review?

Note that this should only include active time on your part. So for example, if you spent 2.5 hours over a week working on an issue, and then get a PR up and request a review, your initial implementation time should be 2.5 hours.

This chunk should include the time you spent:

1. Understanding the issue.
2. Implementing new code.
3. Writing tests or checking your work
4. Getting a PR up for review.
5. Etc.

Post-Review Implementation Time

How long did it take to get the PR ready to merge post-first review?

Note that this also includes active time on your part. So if you get a PR up for review, have to wait three days for a review, and then have to make 20 minutes of changes as a result of the review, the post-review implementation time would be only 20 minutes.

This time bucket might include include:

1. Time spent fixing code because of requested review changes.
2. Time resolving merge conflicts.

NOTE: If you did not get a review on your PR, or if the PR just approved your changes, then this time bucket would be zero minutes!

Perceived Effort We ask you to rate the effort required to solve this issue on a scale of 1-5:

1. Minimal effort: this issue was extremely easy to implement, and required very little effort or concentration. For example: making a simple text change to a webpage, refactoring code following a well-established pattern, copying an existing solution.
2. Below-average effort: this issue was easy to solve, and required less effort than the average issue. For example: creating a new feature with a well-established design, writing unit tests for well-encapsulated functionality.
3. Average effort: this issue required an average amount of effort to implement, and was not notably different from other issues. For example: creating a new feature with some novel components, tracking down a reproducible logic bug.

4. Above-average effort: this effort was hard to solve, and required more effort and concentration than the average issue. For example: Refactoring legacy code with limited tests, implementing complex algorithms or data structures.
5. Maximum effort: this issue was extremely difficult to solve, and required very heavy effort and concentration. For example: re-architecting a major system redesign, debugging critical and complex production bugs with limited information.

G.3 Onboarding call and Cursor Training

All participating developers started the study with a 30 minute introduction and onboarding call. Before the call, developers were asked to set up an account on screen recording software (Loom), install and setup Cursor for their codebase, and read through the data they would be asked to collect over the course of the study.

As all developers had some previous experience with VSCode, developers were all able to setup and use Cursor on their codebases with little overhead. On the onboarding call, developers were given a basic training on Cursor agent mode to ensure they could:

- Create a new agent mode instance on their own codebase.
- Add a relevant file to the context window of the agent.
- Prompt the agent to do make a change to this file.
- Accept changes that the agent suggested to this file.
- Revert changes that they had previous accepted, undoing the agents changes.

Developers were also given a verbal overview of the data they were asked to collect, described in Section G.2.1, and given a chance to ask any questions they had about this data.

G.4 Mid-experiment check-in calls

All developers were offered periodic 15 minute check-in calls to assess their progress, answer any questions they had about the study, and ensure they were on track to complete issues in a timely manner. Most developers had between 1-4 check-in calls over the course of the experiment. These calls also provided an opportunity to ask developers about their experience with using AI at that point in the study.

G.5 Exit Interview

All participating developers were interviewed at the termination of the study, during a 30 minute - 1 hour exit interview. Interview time ranged from 1 day to $\sim$ 6 weeks after developers finished their last issue, depending on available scheduling.

The exit interview was unstructured, and designed to encourage developers to share their qualitative experience during the study. The following outline was followed during the exit interview, but not all questions were asked to all developers, depending on relevance.

Prior Usage:

Collect the prior [AI, Cursor, etc.] usage information we have to confirm it in detail.

Data Audit

Look through their tracked issue data and confirm any data cleanup with them.

Exit interview:

1. On task selection: how did the tasks you worked on compare to the average tasks you do on this open source repository? How were they different?

2. During the study:
   (a) Did you use the same IDE for AI and non-AI tasks? Why?
   (b) Was your experience in this study majorly different from your standard development on this repo? Why?
3. On amount of effort:
   (a) Do you feel using AI or not affected how much effort you used on a given issue?
   (b) How did your level of focus on these issues compare to normal work on this repo?
   (c) How did time tracking or screen recording affect your working?
4. On AI code-cleaning:
   (a) How good did you find the AIs outputs?
   (b) How much cleaning did you do on the AI outputs?
   (c) What is the code quality bar in your repo?
5. On scope-creep:
   (a) Are there any issues that you gave up on because they were harder than you expected and so not worth it?
   (b) Do you feel like the "size" of issues changed as a result of using AI? Specifically, do you feel like the issues were variable sized, and AI pushed you to go bigger?
6. Going forward:
   (a) Do you plan to use AI tools going forward?
   (b) Is this more than you planned to use them before the study?
   (c) Did you increase the amount of AI that you used outside of the study as a result of the study?
   (d) How did the study affect your belief in AI tools?
7. On your AI skill level:
   (a) How confident are you that you use AI effectively now vs. at the start of the study? Do you feel you have improved at using AI?
   (b) Did you notice an improvement in your ability to get useful work from the AI?
      i. What specific strategies worked here?
8. On AI effecting your work:
   (a) Did you find yourself sitting around and waiting on AI to generate code?
   (b) Did you notice a change in idle or distracted time as a result of using AI or not AI?
9. On the effectiveness of AI
   (a) Before the study:
      i. What effect did you think AI tools would have on your time to complete issues?
      ii. What were the primary reasons you thought this?
   (b) During the study?
      i. How much do you believe AI changed your time to complete your issues?
      ii. Specifically: Where did AI seem to speed you up? Where did AI seem to slow you down?
10. Most effective AI tools:
   (a) What AI usage pattern feels the most effective for you?
      i. Cursor vs. a web-browser? Why?
      ii. What model do you prefer, why?

11. Study Experience
    (a) Would you participate in this study again? Why or why not?
    (b) What is one thing you liked about this study, and one thing we could improve?
    (c) Anything else you wished I asked about?

G.5.1 Exit Survey

METR Experiment Exit Interview

1. This form will take you about 15 minutes to complete.
2. Please follow the instructions closely for each question.
3. Do your best to answer accurately.

Thank you for your participation - this is the last step in study participation!

Questions:

1. What is your name?
2. How many hours had you spent using LLMs before the start of this experiment?
   (a) 0 hours
   (b) 1 - 10 hours
   (c) 10 - 100 hours
   (d) 100 - 1000 hours
   (e) $>$ 1000 hours
3. How many hours had you used Cursor before the start of this experiment?
   (a) 0 hours
   (b) 1 - 10 hours
   (c) 10 - 100 hours
   (d) 100 - 1000 hours
   (e) $>$ 1000 hours
4. By the end of this study, how would you rate your skill level at Cursor?
   (a) Very Bad
   (b) Below Average
   (c) Average
   (d) Above average
   (e) Very Good
5. On this repository, I typically make code changes through pull requests. *True/False*.
6. On this repository, I typically check my own code to make sure it's high quality. *True/False*.
7. On this repository, another developer typically reviews my code to ensure high code quality. *True/False*.
8. On this repository, I typically attempt to match repository style guidelines with my contributions. *True/False*.
9. This repository has a high quality bar for code contributions. *True/False*.
10. I typically only submit high quality PRs to this repository. *True/False*.
11. How much did AI decrease or increase the time it took you to complete the issues as part of this experiment?

(a) If using AI resulted in you completing issues 2x faster, put 2.
(b) If using AI resulted in you completing issues 2x slower, put .5 (because 1/2 = .5)
(c) If using AI did not change how long it took you to complete issues, put 1.

12. During this study, what best describes how you read AI generated code that you included as part of your implementation?
    (a) I don't read AI generated code I use. I just check if it's outputs are correct.
    (b) I typically skim AI generated code I use to see if it's correct.
    (c) I typically read every line of AI generated code I use to check it's correct.
13. During this study, what best describes how you edit AI generated code that you used as part of your implementation?
    (a) I usually take AI code as-is, without making edits.
    (b) I usually make minor changes to AI generated code (like deleting comments or changing formatting).
    (c) I usually make major changes to AI generated code (like deleting pieces of code, adding new features, or refactoring code)

G.6 Participant Dropout

Over the course of the study, we stopped collecting work from three developers. Two of them were because the repository they contributed to paused development indefinitely, and the third developer was due to widespread cheating in the first set of issues they contributed. These developers were compensated fully for their work as part of the study, and we exclude their issues from all results.

G.7 Developer and Repository Statistics

Repository names and descriptions for repositories for which developers did not give consent to share their names are redacted.

Dev	Repository	Months Since First Commit	Commit Count	Commit Rank	AI-allowed Issues	AI-disallowed Issues
2	mito-ds/mito	30	3000	3/30	13	11
3	stdlib-js/stdlib	300	30000	3/300	9	12
4	ghc/ghc	30	300	30/3000	8	12
5	haskell/cabal	30	30	30/300	11	8
6	stdlib-js/stdlib	30	300	3/300	11	7
7	flairNLP/flair	30	3000	3/300	12	5
8	jsdom/jsdom	300	300	3/300	8	9
9	HypothesisWorks/hypothesis	30	300	3/300	11	6
10	devflowinc/trieve	30	300	3/30	10	5
11	scikit-learn/scikit-learn	30	300	30/3000	4	7
13	EleutherAI/gpt-neox	30	30	3/300	5	5
16	huggingface/transformers	30	300	3/3000	1	1
1	Anonymized	300	3000	3/300	15	13
12	Anonymized	30	3000	30/300	9	2
14	Anonymized	30	300	3/30	4	4
15	Anonymized	30	300	3/30	5	3

Table 7: *Maintainer statistics for the study participants, sorted by total number of issues. The table shows representative values (nearest to 3×10^x) and percentages rounded to nearest bucket (10%, 30%, 50%, 70%, 90%) to preserve anonymity while maintaining relative scale.*

Repository	Stars	Forks	Committers	LoC	Age (years)	AI-allowed Issues	AI-disallowed Issues
stdlib-js/stdlib	5264	843	128	8M	9	20	19
mito-ds/mito	2468	180	10	700k	3	13	11
ghc/ghc	3134	720	1008	1M	19	8	12
haskell/cabal	1676	714	532	300k	21	11	8
flairNLP/flair	14213	2119	278	60k	7	12	5
jsdom/jsdom	21082	1738	350	1M	15	8	9
HypothesisWorks/hypothesis	7910	616	355	100k	12	11	6
devflowinc/trieve	2343	203	68	800k	2	10	5
scikit-learn/scikit-learn	62566	26025	3164	400k	15	4	7
EleutherAI/gpt-neox	7251	1068	132	100k	4	5	5
huggingface/transformers	146580	29562	2956	2M	6	1	1
Anonymized	30000	3000	300	300k	30	15	13
Anonymized	300	30	30	30k	3	9	7
Anonymized	300	300	300	300k	30	9	2

Table 8: *Repository statistics for the study, sorted by total number of issues. The table shows representative values (nearest to 3×10^x) for anonymized repositories.*

G.8 Screen Recordings

The screen-recording labeling process is time-intensive. As a result, screen recording labeling was started early in the data collection process, and to maximize the number of fully-labeled recordings, shorter recordings were prioritized first. Additionally, many developers choose to not record their screen if they were making a small set of changes due to a review. These factors may bias estimates of time allocation.

The following instructions were given to coordinate labeling screen recordings with fine-grained activity labels.

Overview

1. As part of an experiment we're running, we're labeling the loom videos that developers recorded of them implementing PRs on large open source repositories.
2. The goal of this labeling is to understand how these developers actually spend their time when they are programming – so the labels include things like "writing_code" or "reading_code" or "reading_docs."
3. A very important piece for us to understand is how they use and interact with AI. This practically means special labels around their use of Cursor Composer / Agent Mode.

Requirements: You're a good fit for labeling this data if:

1. You know how to program well, and when watching someone program over their shoulder can figure out what they are working on.
2. You have used Cursor Composer before, and know how that works!

Compensation

1. We'll pay standard per hour rates for image labeling.
2. We'll be checking 1/10 of the submissions. If your timing labels are sufficiently accurate (close to our hand-checked solutions), we'll give you a $250 bonus.

How to Label:

1. First, scroll down and read the labels below. Feel free to leave comments if you have any questions about these.

2. Then, open the tracking sheet:
   (a) Claim one of the unclaimed videos in the "To Label" sheet by putting your name in one of the columns. Go in-order, so we get shorter videos first.
   (b) Then, make a new tab, and copy over the 'Template' tab. Name the new tab as the initials of the person who made the recording, followed by a dash, and then the issue_id number.
3. Open the loom video link:
   (a) You probably need access to METR's loom account for this; if you do not have access, please ask us and we'll add you!
   (b) If you have access to METR's loom account but do not have access to the particular loom video you opened from the sheet, please do not request access. Just mark this in the sheet, and move on to the next video.
4. Take notes using this tool. It makes Loom note taking much easier, and means you don't have to leave the loom page!
   (a) The default rate is 5, but you can adjust this with 'rate 2' to make it slower.
   (b) You can drag this note taking app around the screen, and it works in fullscreen mode. Click instructions at the bottom to see more commands!
   (c) Note: please try and make the start and end time of your notes correspond to the actual start and end times of the things users are doing. This might require rewinding the video!
5. After you're done watching the video and taking notes, type 'done' and copy the results into the sheet.
6. Note: if the video is >20 minutes long, copy your notes out in 20 minute chunks, to make sure that your labels are as accurate as possible.
7. Then, go through the 'Label' column of the sheet, and label each chunk of time with the columns below.
8. Then, go through the 'AI Use Label' and 'AI Type Label' column and label any of the AI usage with what the developer is using the AI for as well as the model/UI being used.

The Labels

Label accuracy is very important for this data work. As such, it requires a fair bit of critical thought about what the user is really engaged in.

If you're not sure what the user is doing, please put "unknown" as the label. We can always go back and fill things in, but only if you note this. You can also leave a comment to the side of the row describing what's confusing to you!

If you think that there's a better label for things than one provided, feel free to add it + tag me in a comment on top of it. I can then add it to the list here :)

The Labels Most Common

1. reading_issue: the dev is reading the issue that they are planning to implement a fix for as a part of this loom video.
2. writing_code: the dev is actively writing code. They might also be reading a bit or navigating around, but mostly they are editing/writing code on the page. (Note that writing testing code is counted differently).
3. reading_code: the dev is primarily reading existing code. They might be navigating through the codebase to find specific things, but in practice they are reading.
4. reading_docs: the dev is reading documentation. Potentially of their own codebase, potentially of some other codebase/tool. It's not code they are reading.

5. writing_docs: the users are writing documentation. This could include release notes or a documentation page.
6. writing_tests: the user is writing testing code, rather than writing some other type of code.
7. test_running_tests: is actually running the testing code, and looking at the results.
8. test_running_ci: is running or waiting on CI checks that are running on Github/Gitlab.
9. test_manually_checking: testing some solution, but doing this by hand (either by writing some code, or looking at some artifact/output).
10. replicating_bug: replicating a bug, normally the bug described in the initial issue.
11. running_debugger: if the user running the debugger, then note this here.
12. compiling: waiting on some code to compile.
13. setup: running some setup process (e.g. opening their IDE, installing extensions, etc).

Git Related Things:

1. branching: creating a new branch, and adding on to it.
2. committing: adding files or writing a commit message. Some folks try hard on these!
3. pr: getting up a PR and potentially writing a PR message.
4. git: some other misc. git operation (e.g. if they are viewing Git diffs, or something).

Misc:

1. thinking: the user is not AFK, but appears to be thinking through what they are going to do next.
2. unrelated: the user is doing something unrelated, like watching a youtube video or changing their music.
3. paused: the user appears to have stepped away from their computer.
4. communicating_with_teammates: for example, the user switches to slack or discord and asks a question.
5. broken: something is wrong with the loom video.

On using AI tooling

Note: it can be a bit hard to label AI generated usage. The key details here are to describe the full flow of how the user writes a prompt, waits, and then either accepts or rejects it – and then what they do after.

For a good example of what this should look like, see this labeling.

1. writing_prompt: the user is writing a composer prompt.
2. waiting_on_generation: the user is waiting on the AI to generate code or a response.
3. reading_generation: the user is spending time actually reviewing the suggestions the AI has made.
4. generation_taken: the user takes the suggestion from the AI, and it turns out to be useful / they don't ditch it in the future.
5. generation_rejected: the user does not take the suggestion from the AI, or takes the suggestion and then reverts back to before they took the suggestion. In other words, they don't use the AI generated code because it's broken.

6. ai_code_cleaning: if the user takes a suggestion from the AI, and then spends time cleaning that code, then we label this not as writing_code but instead as ai_code_cleaning. This includes changing spacing, minor refactors, etc. As long as the users keeps the bulk of the code, this is considered a generation_taken.

Feel free to also use 'ai_docs_cleaning' or 'ai_commit_cleaning' if this is what the user is cleaning up.

AI Use Label: We also ask that you fill out the AI use label column to describe

1. new_feature: the user is using AI to extend functionality of the codebase.
2. bug_fix: the user is prompting the AI to fix a bug in the existing codebase.
3. code_search: the user is using the AI to search their codebase for some code / implementation detail.
4. tests: the user is having the AI generate code for testing reasons.
5. docs: the user is using the AI to write docs. Could include readme, or git commit messages.
6. question: the user has a question (e.g. one they could ask google) that they are adding here.
7. integration: the user is integrating some code with an external system, and so the code-gen is primarily for understanding or integrating with that system.
8. refactoring: improving code, without extending the code's functionality or fixing bugs

NOTE: if you think there are other composer usages that this better fits into: please feel free to just write what you think best describes what the user is doing here!

AI Labels: We also ask that you mark three columns that describe where / how the user is using AI:

1. AI Model Label: The model being used E.g. "3.7 Sonnet" "3.5 sonnet" "o1" "o1-preview" "gpt4.5"
2. AI UI Label: The UI being used E.g. "cursor composer" "cursor chat" "web UI" (the respective LLM providers' chat website)

This should co-exist with the AI Use Label above. So, for example, a segment of video in which someone is writing/reading cursor compose might have AI Use Label "new_feature" and AI Type Label "Sonnet 3.7, cursor composer"

FAQ:

1. *How accurate do the timestamps need to be?* Roughly correct. It's ok if the timestamps are off by a few seconds on each end, but in general you should try and avoid large (e.g. 10+ second) errors.
2. *When is a user writing code vs. reading code?* These are often interleaved. In practice, if the user is writing code for $>$50% of the chunk of time, this is writing code – only if they are reading code for like $>$30 seconds is it really like a concrete "reading_code" time.
3. *I have never used Cursor composer.* You're probably not a great fit for this labeling, in this case.
4. *I am not sure how to label things.* If you are confused about how things should be labeled (e.g. there's some weird cursor flow you don't understand where a user accepts code changes, and then later reverts), just label things as "unknown" and we can come back to it.

G.9 Instructions Given to Expert Forecasters

Supplementary Information for METR AI speedup study survey

METR is currently running a field experiment measuring how AI tools impact open source developer productivity.

The TL;DR is that we recruit experienced developers who contribute to popular open source projects, randomize their tasks to having no AI or AI allowed, and measure the ratio between the time it takes a human to complete tasks with AI vs. without AI. The study aims to measure speedup in conditions that closely mirror normal software development.

In this supplementary information document, we first describe two pieces of relevant background: the structure of open source software development, and AI tooling. (If you are highly familiar with open source software development or cursor agent mode you should probably skip the respective sections.) We then describe the experiment in more detail: how we sampled developers and repositories, the tasks developers work on, how developers participate in the study, and finally how we intend to estimate speedup due to AI.

(We are only part-way through running the study, so do not yet know the final result ourselves.)

Background

[Section F.2 was then included.]

Experiment

Contributor recruitment

Open-source contributors were recruited through a multi-stage process to select for active contributors to repositories that had more than 500 stars.

1. Initial outreach was conducted via professional networks, ML-focused communities (Reddit's r/Python, r/MachineLearning), and through GitHub profiles.
   (a) GitHub profiles were found by searching GitHub for the 250 most popular repositories, as well as those tagged with: ai, llm, deep-learning, neural-networks.
   (b) Contributors to these repositories were filtered to exclude those who had committed less than five times in the previous three months, and then emailed.
2. Interested contributors (n=50) filled out a preliminary survey to assess:
   (a) Years of software development experience
   (b) The repositories they contribute to

All contributors who planned to contribute to repositories with more than 500 stars were offered an introductory call to provide an overview of the study timeline and parameters. 31 calls were conducted, with half of developers being filtered out for a lack of previous contribution experience or because the timeline didn't work.

The remaining 16 developers were then given access to Cursor Pro. We had a 30-minute call with each developer where we set them up with a data collection template, answered questions, and trained them on Cursor. Developers were considered trained once they could use Cursor agent mode to prompt, accept, and revert changes to a file on their own repository. 94% of developers noted that they had used web-based LLMs as part of their development workflow before participating in our experiment. Rates of past usage of Visual Studio Code, Github Copilot, and Cursor are 63%, 56%, and 25% respectively [note: these were preliminary numbers, and are lower than the true values reported in the paper].

All participating repositories are listed below.

[Table 8 was then included.]

Issues

Each contributor maintained a list of issues to work on as part of this study. Contributors were asked to select issues as they would during normal development on this study, with the caveat that they should break issues that were likely to take > 4 hours into sub-issues that take ≥ 2 hours if possible.

The issues are intended to be as similar as possible to those that would have been worked on if this study never took place.

After collecting this issue list, each issue was randomized to either AI-allowed or AI-disallowed conditions. If AI is allowed, developers can use any AI tools they so choose, including no AI tooling if they deem it not helpful to the problem. If AI is disallowed, no generative AI tooling can be used.

Study Participation

Contributors completed issues much as they would outside of our experiment, with two important differences: they record their screen as they work, and they take implementation notes post issue completion. (We use human-labelled video recordings covering the majority of issues to confirm compliance.)

For the duration of the study, we periodically check in with developers and provide feedback on their implementation notes and loom videos. We occasionally emailed developers with tips on how to use Cursor more effectively if we notice some easy wins in their Loom videos.

Measuring speedup

We aim to measure the speedup factor due to AI, defined as:

S = mean(completion time with no AI) / mean(completion time with AI allowed).

S = 2 would indicate issues assigned to AI allowed taking half the time of issues assigned to no AI (100% speedup); S = 1 would indicate that issues take the same time to complete with and without AI being allowed (0% speedup); S = 0.5 would indicate that issues assigned to AI allowed take twice the time of issues assigned to no AI (-50% speedup).

(We are asking you to predict S, i.e. the quantity taking value 2/1/0.5 rather than 100%/0%/-50% in the examples.)