Who is using AI to code?
Global diffusion and impact of generative AI

Simone Daniotti*[1,2], Johannes Wachs[2,3,4], Xiangnan Feng[2], and Frank Neffke[2]

[1]University of Utrecht

[2]Complexity Science Hub

[3]Corvinus University of Budapest

[4]HUN-REN Centre for Economic and Regional Studies

Generative coding tools promise big productivity gains, but uneven uptake could widen skill and income gaps. We train a neural classifier to spot AI-generated Python functions in 80 million GitHub commits (2018–2024) by 200,000 developers and track how fast—and where—these tools take hold. By December 2024, AI wrote an estimated 30.1% of Python functions from U.S. contributors, versus 24.3% in Germany, 23.2% in France, 21.6% in India, 15.4% in Russia and 11.7% in China. Newer GitHub users use AI more than veterans, while male and female developers adopt at similar rates. Within-developer fixed-effects models show that moving to 30% AI use raises quarterly commits by 2.4%. Coupling this effect with occupational task and wage data puts the annual value of AI-assisted coding in the United States at \$9.6–\$14.4 billion, rising to \$64–\$96 billion if we assume higher estimates of productivity effects reported by randomized control trials. Moreover, generative AI prompts learning and innovation, leading to increases in the number of new libraries and library combinations that programmers use. In short, AI usage is already widespread but highly uneven, and the intensity of use, not only access, drives measurable gains in output and exploration.

*Corresponding author: `daniotti.simone@gmail.com`

According to proponents, Artificial Intelligence (AI)—in particular Large Language Models (LLMs) and generative AI—will drastically increase our productivity and revolutionize the way we work. For instance, generative AI is expected to complement or substitute humans in an increasing set of tasks (*1*). This forces individuals, firms, and policymakers to make important decisions about the use and regulation of generative AI under major uncertainty. The stakes are high: generative AI has become widely accessible through tools such as ChatGPT or Claude, directly complements human thinking (*2*), and holds the potential of becoming a general-purpose technology that can solve a wide variety of problems (*3*).

Experimental and quasi-experimental evidence so far supports the notion that generative AI has transformative potential, showing that Generative AI leads to increases in productivity and output of individual workers in a variety of jobs (*1, 4, 5, 6*). Surveys and data reported by LLM owners suggest that these technologies are diffusing rapidly (*7, 8, 9*). Yet, estimates of the aggregate impact of AI on GDP and employment are often modest (*10, 11*). Resolving such contradictions requires accurately determining adoption rates, intensity of use, and productivity effects at a global level. However, surveys are often limited to a single country and their self-reported adoption rates may be inaccurate (*12*) or biased (*13*). Randomized controlled trials (RCTs) (*14, 6, 1, 4, 5*) or natural experiments (*15, 16, 17*) consider an individual treated if they have access to generative AI tools, leaving intensity of use or importance in real work activities unobserved. Moreover, both surveys and experiments tend to observe individuals over short time periods, obscuring any learning effects. To address these challenges, we quantify the diffusion and effects of generative AI by measuring its use at a fine grained level in software development, an important and high-value sector (*18, 19*) that is uniquely exposed to generative AI (*20, 15, 14, 21*).

Although the general diffusion of generative AI is undisputed, self-reported adoption rates differ markedly across demographics, seniority, work experience, and sectors (*9, 8*). Evidence from job ads and firm websites suggests that adoption of generative AI varies across geography (*22, 23*), possibly due to economic, cultural or regulatory differences (*24, 25*). If generative AI indeed substantially raises productivity, any implied barriers to adoption will have significant consequences for inequality within and across countries (*26*). However, historically, macro-level productivity effects of general-purpose technologies have taken long to materialize (*27, 4, 28, 29*). Together, this leads to substantial uncertainty about the impact of generative AI today.

Our research addresses this uncertainty with a large-scale empirical study of generative AI usage in software development. To do so, we design and implement a machine learning classifier to identify code written with substantial AI assistance in over 80 million software developer contributions to open-source Python projects on GitHub. This allows us to analyze shifting patterns of AI-generated code at a highly granular level. To train this classifier, we assemble a custom training set, combining existing sources with a procedure that generates synthetic training data. We then apply the classifier to contributions to GitHub projects between 2019 and 2024. Linking commits to developers' locations, this allows us to trace the timing and intensity of AI adoption across countries.

We observe noticeable growth spikes in AI-generated code soon after key generative AI releases such as GitHub Copilot, the original ChatGPT, and GPT-4.0, highlighting how breakthroughs in LLMs prompt rapid uptake by developers worldwide. Yet, we also observe significant differences among countries. In the US, which leads in the intensity of use of AI throughout our data, AI-written code grows from 0% in 2020 to approximately 30% by the end of 2024. Contributions from Germany, France, India, Russia, and China vary between 12 and 24% in 2024, suggesting significant geographic barriers to diffusion. Furthermore, corroborating existing studies (*8, 9*), adoption rates are higher among less experienced programmers. This would, other things equal, compress productivity inequality between junior and senior workers. Unlike most previous work, we find no significant differences between men and women.

AI use changes user activity. Models that exploit changes in AI usage by the same developer at different points in their career suggest that use of generative AI increases programming contributions. Based on our measurement that by the end of 2024 AI writes 30% of functions contributed by US developers, we conservatively estimate that generative AI increased the volume of such contributions by 2.4%. Combining wage information with estimates of how much programming happens in each of nearly 900 US occupations, this estimate implies that generative AI generates \$9.6-\$14.4 billion dollars in value annually just in the US software sector by 2024. Pooling the larger productivity effects reported in recent RCTs and natural-experiment studies (*30, 6, 15, 17*) drives the implied annual value of generative AI in software well beyond this conservative baseline.

Finally, AI-use also is associated with changes in the nature of the code developers produce, in particular, in terms of the exploration of new software libraries and combinations of such libraries.

Taking libraries as an expression of the type of functionality developers try to program, this finding suggests that generative AI helps developers extend their capabilities, and to learn to move into new programming areas.

Results

Figure 1 illustrates how we classify code contributions as either human or AI generated. We train a machine learning classifier on a dataset containing examples of Python code written by humans and by AI. To focus on coherent units of code, we limit this analysis to blocks of code that represent functions. This choice allows us to focus on a fine grained elemental unit of software development. Following (*31*), we transform these functions using *GraphCodeBert*—a pre-trained language model for code that embeds a function into a high dimensional vector space using its tokens, comments, and the dataflow graph of its variables (*32*)—that is fed into a classifier to determine whether a given function was written by a human or by generative AI.

When training the classifier, we simultaneously retrain the embedding weights to optimize results. To do so, we construct a ground truth dataset. We start by collecting Python functions of which we are certain they were written by a human programmer. First, we take functions written in 2018, as they predate the release of capable generative AI models. Because programming styles evolve over time, we add functions created in later years from the HumanEval datasets for the years 2022 and 2024. Second, to generate a dataset of similar functions that we know were written by generative AI we apply a two step procedure. For each human-written function, we ask one LLM to describe the function in English, specifying the type of input and output of the function (similar to modern agents' AI tools). Next, we feed this text to a second LLM and request the model to generate a function based on this description. The result is an AI-generated function that is close in functionality to the original human-written function. Previous approaches attempt to use the same instance of an LLM to both describe and generate the function, creating unneeded strong correlations between the human code and its transcription (*33*). Our approach, detailed in the Materials and Methods section, ensures that our training corpus consists of closely matched human and AI generated functions.

The classifier performs remarkably well, achieving an out-of-sample ROC AUC Score of 0.964

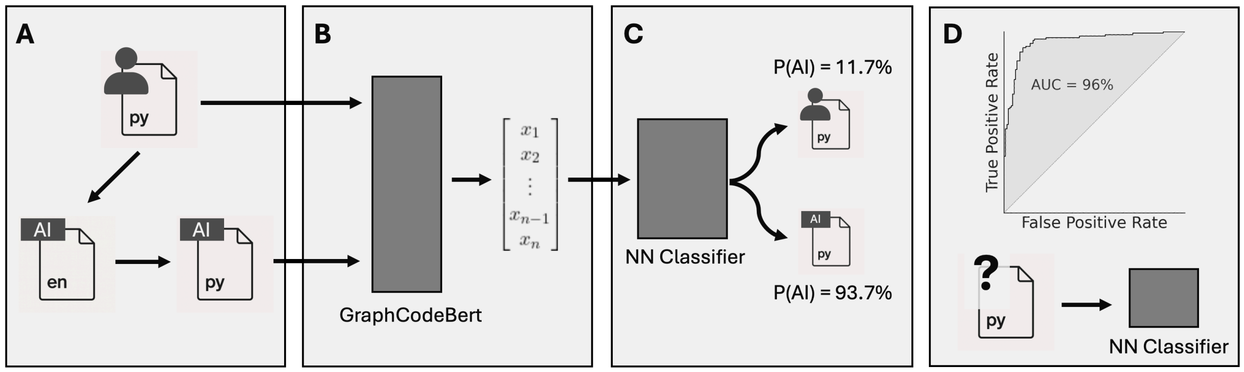


Figure 1: Classifying code from functions written in the Python programming language as human or AI generated. A) Using a collection of human generated code, we ask one LLM to describe the code in English, then another to implement that description as a Python function. B) We vectorize the resulting code using GraphCodeBert, an embedding method that uses a code's tokens, comments, and variable graph flow. C) We train a neural network classifier combining GraphCodeBert with a classification head to predict the human/AI labels. D) We evaluate the classifier on out-of-sample data and apply it to a large database of unlabeled Python functions.

and Average Precision of 0.969. We apply this classifier to 31 million functions extracted from 80 million contributions to Python projects from the beginning of 2019 to the end of 2024 for the full population of US-based users and a quarterly sample of 2,000 users in five other countries: Germany, France, India, China, and Russia.

Figure 2 plots the AI adoption trajectory for US developers. Adoption rates show sharp increases following major AI advancements, including the launches of Copilot, ChatGPT, and second generation LLMs. Panel B compares the US to five other major countries in the global race toward AI adoption, randomly sampling commits from software developers in these countries. This shows that United States took an early lead, which it has managed to maintain ever since. Over 30% of Python functions in the US were generated by AI by the end of 2024, closely followed by 23/24% for Germany and France. India reaches comparable levels at 20%, after having initially lagged in adoption. In contrast, Russia and China have so far remained late adopters.

Focusing on US developers, whose complete set of Python commits were analyzed, we find that AI adoption rates drop with the number of years that developers have been active on GitHub. Fig. 3b shows that whereas the most experienced developers use generative AI in 28% of their code, programmers who have just joined the GitHub platform use these tools for 41% of code. In contrast, using (self-reported) first-name-based gender-projection algorithms, we find no difference

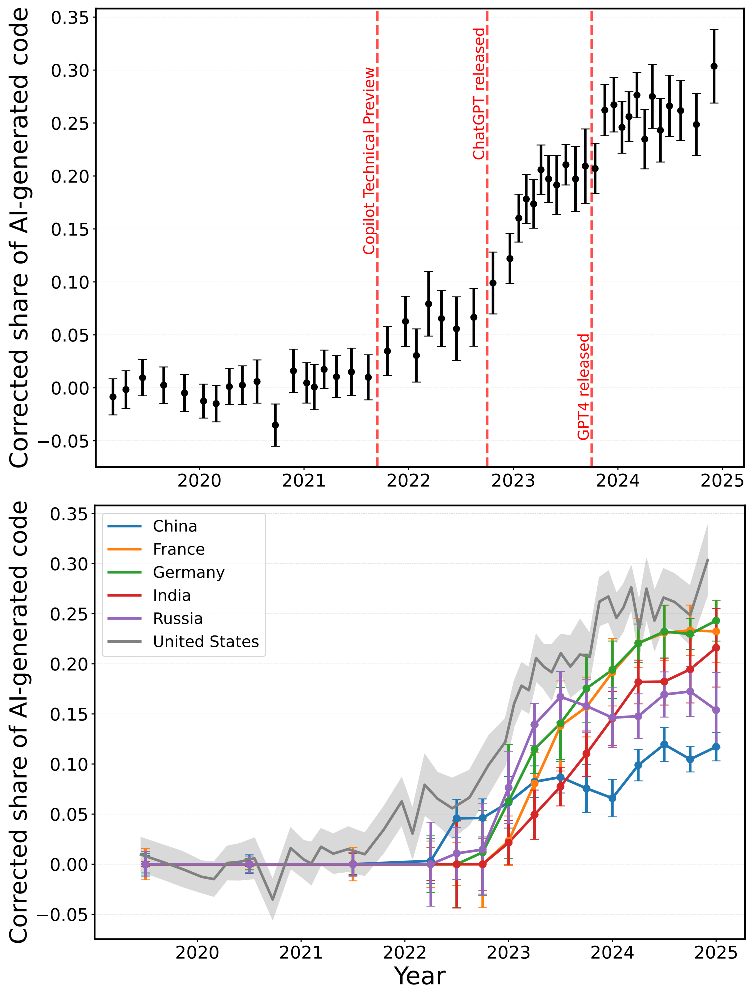


Figure 2: **Share of AI-generated Python functions over time. A**: share of Python functions that were created or substantially changed by GitHub users in the United States. Vertical lines depict 95% confidence intervals. The plot reveals abrupt shifts in adoption coinciding with key AI-related events: the release of GitHub Copilot Preview, the public launch of ChatGPT, and the second wave of LLM releases (GPT4 and related models). **B**: adoption in China, France, Germany, India and Russia. Each marker represents a sample of 2,000 programmers. The US curve is replicated from panel A as point of reference. The US lead the early adoption of generative AI, followed by European nations such as France and Germany. From 2023 on, India rapidly catches up, whereas adoption in China and Russia progresses more slowly

between men and women (Fig. 3a).

To assess how generative AI impacts the quantity and nature of code that programmers produce, we rely on regression models with user and quarter-of-year fixed effects. This compares the output of the same programmer at different points of AI adoption, controlling for economy-wide trends. These models, summarized in Fig. 3c, suggest a substantial impact of generative AI. At the average

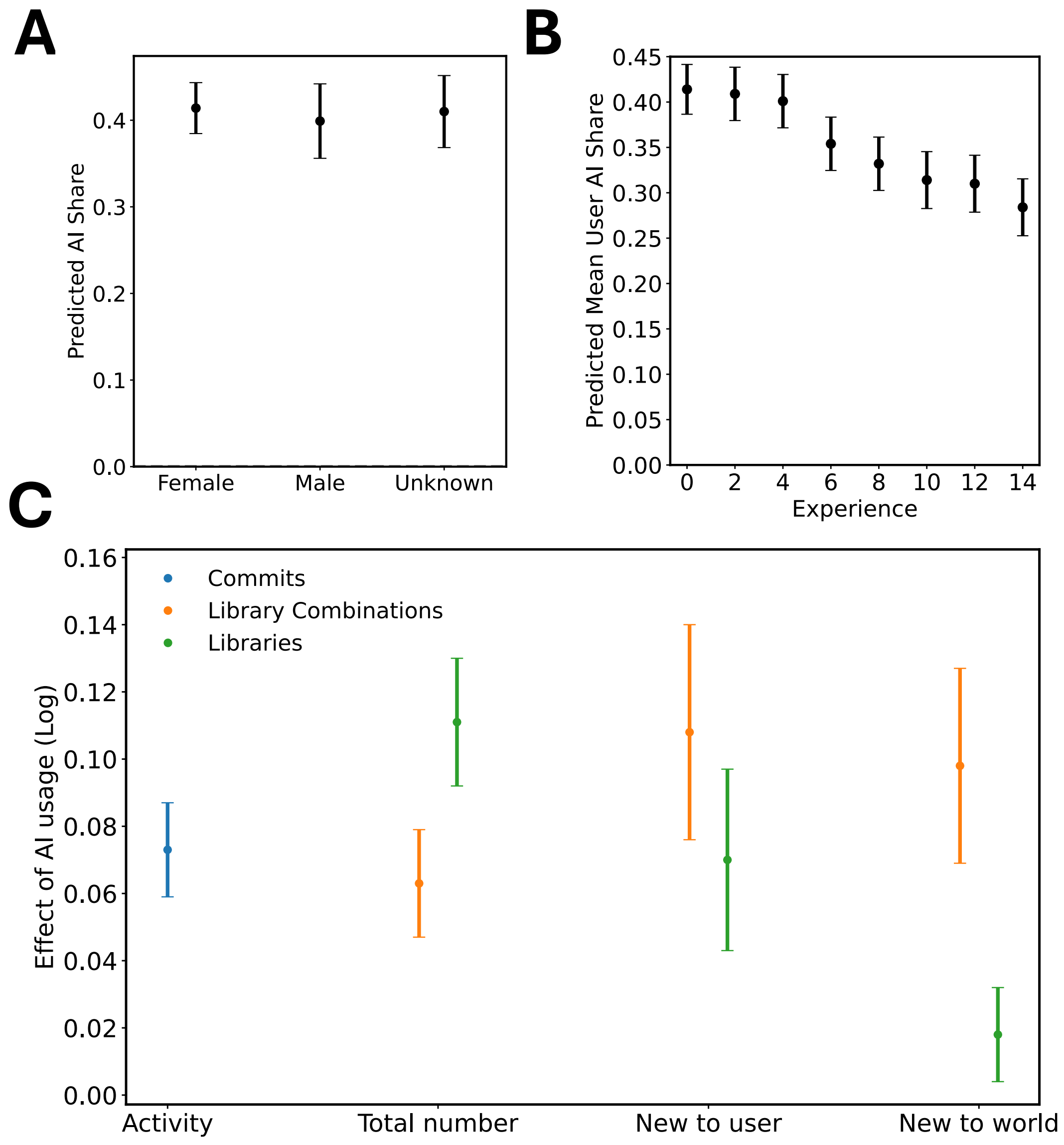


Figure 3: A) Intensity of AI use by gender inferred from GitHub display names, US data 2024. B) Intensity of AI use by user tenure on GitHub, US, 2024. C) The estimated effect of AI use on user activity, derived from a user-quarter panel regression model with user and quarter fixed-effects. Using more AI is associated with increased commit activity, greater usage of libraries and combinations of libraries, and an increased likelihood that users use a new (to the user, or globally) library or combination of libraries. In all figures, error bars indicate 95% confidence intervals.

US adoption rate for the end of 2024, generative AI use accounts for 2.4% higher output (measured in terms of the number of commits).

We check the validity of our estimates by carrying out placebo tests. These show that in the years before the introduction of generative AI coding tools—where any variation in AI usage reflects false positives—have no explanatory power for any of the output variables described in Fig. S3. Moreover, in Fig. S4 of the SI we show that measurement error likely conservatively biases the estimates of the effects of AI downwards.

To assess the economic significance of the estimated productivity effects, we conduct a back-of-the-envelope calculation of the value of generative AI in programming to the US economy. To do so, we first estimate how much money the US spends on coding tasks in terms of wages. Combining detailed information about tasks performed in almost 900 different occupations with estimates of average annual earnings in these occupations, we put this at between \$400 and \$600 billion a year (see Materials and Methods). This values productivity effects of generative AI at between \$9.6-\$14.4 billion per year for coding activities in the US alone. Note that these calculations do not account for any potential reduction in value of coding tasks due to additional supply of code by AI, assume that effects in Python programming are representative of effects in other languages and that open source projects on GitHub have similar AI usage rates as in walled-off programming projects inside companies.

Yet, this estimate almost certainly understates the true economic impact of generative AI in software development. Field experiments that track developers in live codebases report increases of roughly 13.6% in commits and a 26% faster task completion rate (*6*). Natural experiments exploiting the staggered roll-out of GitHub Copilot find that access to Copilot leads to a 6% increase in merged pull-requests (*17*) and a 5.5% increase in commits (*15*). In lab experiments, subjects with access to generative AI complete software tasks 21-55% faster, translating to a 6.3%-16.5% effect assuming a 30% AI use rate (*14, 30*). Combining the average of the three estimates of improvement in task-completion time from RCTs (16.5% (*14*), 6.3% (*30*), and 26% (*6*)) with our estimated 30% AI usage would imply an annual value of \$64-\$96 billion. Relying instead on the average increased commit and merged pull request rates across one RCT and two natural experiments (13.6% (*6*), 5.5% (*15*), and 5.6% (*17*)) yields an 8.2% increase in productivity or \$33-\$49 billion in annual value.

Apart from increasing activity rates, AI adoption is also associated with increased experimentation with new libraries and combinations of libraries. The average usage rates by US developers of 30% imply that users introduce an additional 2.2% new libraries and 3.5% library combinations. Because different libraries allow for different types of functionality in a software scripts — from numerical methods using *numpy*, to data visualization with *matplotlib*, and front-end web development using *django* — we follow (*34*) and interpret this as a sign that programmers expand the range of coding into different areas of software development.

Discussion

The nature of work often changes with the introduction of new technologies. Understanding these changes is especially difficult when the innovation in question is radical (*35*), and at the same time pervasive (*36*). The uncertainty of the effects of generative AI on work and labor markets is reflected in the wide range of attitudes researchers and policymakers take towards it, ranging from utopian to skeptical or outright apocalyptic. These attitudes are formed in a fast-moving context, and are based on incomplete evidence on the adoption and effects of AI. The findings in this study provide better evidence of how generative AI is used in a large, important, and highly exposed sector of the economy, as well as a way to monitor this in real-time going forward. Applying an AI detection classifier to millions of code contributions made over a five-year period, we can confirm that AI adoption is fast, but heterogeneous world-wide and across individuals. We likewise show that AI adoption increases individual coders' activity and the breadth of their coding toolkit.

Our results on the productivity effects and heterogeneous diffusion of generative AI raise important questions for policymakers and researchers. We need to understand the barriers of adoption to AI: are these similar to prior radical innovations or is this time different? Moreover, these barriers need to be understood not only at the individual level, but also at the firm, regional, and national levels. Our study takes a first step toward answering such questions.

Indeed, whereas existing literature typically focuses on access to generative AI or usage in controlled experimental settings, our approach allowed us to quantify the intensity with which the technology is used in actual work activities. That said, several limitations of our approach should be noted. First, our analysis focuses on software development. Although this limits its scope, work in this sector is uniquely amenable to quantitative analysis at a level or granularity that is required to study how AI affects workers and their tasks. Within software, we focus only on Python-based open-source contributions. While Python is a widely used language, adoption patterns may differ in other programming ecosystems. However, our focus on GitHub projects may distort estimates for diffusion rates in China, where the programming community also relies on an alternative collaboration platform, Gitee (*37*). Second, our geographic analysis is constrained to a subset of countries and it would be important to widen the analysis to include countries at different levels of development. Moreover, given the wide dispersion in productivity across programmers (*38, 39, 40*), future research should explore how AI adoption affects not only broad developer activity but also at the upper tail of elite programmers, where the most significant breakthroughs and innovations are likely to occur. Finally, our study exclusively focused on programming tasks. Yet, one study of elite software developers suggests that access to generative AI leads to a shift from managerial tasks to coding (*15*), suggesting that an important margin along which productivity effects may materialize is shifts in the task composition of software developer jobs.

In general, our estimates of the most recent adoption rates in the US of around 30% are remarkably similar to adoption rates claimed for coding work at Microsoft [1] and at Google [2]. Moreover, despite our focus on the software industry and code from open source Python libraries specifically, our results closely align with estimates of adoption and effect sizes from other contexts. For instance, it is widely accepted that older and more experienced individuals are less likely to try generative AI (*8, 41*). However, while most previous studies find differences in generative AI adoption across genders, we do not observe any such difference in our data.

Unlike most other studies, our design allows us to compare adoption rates across countries. Here, we find a clear and sustained lead by US programmers. This may be due to regulatory differences among countries (*24, 25*) or sanctions. Although sanctions are not a strong barrier to LLM use by individuals (*42*), they may have more salient effects on coding inside firms. However, other major countries are quickly catching up, eroding the US' first-mover advantage, at least when it comes to the use of generative AI.

Despite controlling for observed and unobserved user characteristics and performing placebo tests, our estimation design is not optimized for identifying causal effects. When it comes to the productivity effects of generative AI, our estimates are generally smaller than those found in RCTs (*30, 6*) and studies exploiting natural experiments (*15, 17*). However, combining the results on adoption and productivity effects, our conservative estimates still indicate substantial costs savings in coding due to generative AI, ranging from \$9.6 to 14.4 billion in the US in 2024 alone. Relying

[1]https://cnb.cx/3YorqDQ

[2]https://www.nytimes.com/2025/05/25/business/amazon-ai-coders.html

instead on the estimated impact of generative AI on productivity in software by studies that focus on causal effect estimation yields estimates as large as $64 to 96 billion.

Given that generative AI has diffused quickly beyond the US, global cost savings would be substantially larger, even if we confine ourselves to the software sector. Moreover, we are currently still in the early phases of the diffusion curve of what looks to be a new general purpose technology. Historically, early-stage productivity effects of general purpose technologies have been hard to identify because it takes time to integrate them into firm level workflows and procedures, train individuals and amass the complementary assets needed to fully exploit their potential. Based on this, we find ourselves on the bullish side of the debate when it comes to the productivity effects of generative AI.

One of the most surprising findings, however, is the fact that generative AI increases experimentation with new libraries, suggesting that generative AI allows users to advance faster to new areas of programming, embedding new types of functionality in their code. This corroborates prior findings (*43*) that generative AI increases individual innovation, but contradicts this work's claim that this comes at the cost of collective innovation. Instead, we find that generative AI pushes both the individual's own frontier and the global frontier in terms of the use of new combinations of libraries, which has important consequences for innovation in software and for how programmers learn in the presence of generative AI.

References and Notes

1. F. Dell'Acqua, *et al.*, Navigating the jagged technological frontier: Field experimental evidence of the effects of AI on knowledge worker productivity and quality. *Harvard Business School Technology & Operations Mgt. Unit Working Paper* (24-013) (2023).

2. E. Mollick, *Co-intelligence: Living and working with AI* (Penguin) (2024).

3. T. Eloundou, S. Manning, P. Mishkin, D. Rock, GPTs are GPTs: Labor market impact potential of LLMs. *Science* **384** (6702), 1306–1308 (2024).

4. E. Brynjolfsson, The productivity paradox of information technology. *Communications of the ACM* **36** (12), 66–77 (1993).

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

6. Z. K. Cui, *et al.*, The Effects of Generative AI on High-Skilled Work: Evidence from Three Field Experiments with Software Developers (2024), doi:10.2139/ssrn.4945566.

7. T. Teubner, C. M. Flath, C. Weinhardt, W. Van Der Aalst, O. Hinz, Welcome to the era of chatgpt et al. the prospects of large language models. *Business & Information Systems Engineering* **65** (2), 95–101 (2023).

8. A. Bick, A. Blandin, D. J. Deming, *The rapid adoption of generative ai*, Tech. rep., National Bureau of Economic Research (2025).

9. A. Humlum, E. Vestergaard, The unequal adoption of ChatGPT exacerbates existing inequalities among workers. *Proceedings of the National Academy of Sciences* **122** (1) (2025).

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

11. A. Humlum, E. Vestergaard, Large Language Models, Small Labor Market Effects. *University of Chicago, Becker Friedman Institute for Economics Working Paper* (2025-56) (2025).

12. F. T. Juster, F. P. Stafford, The allocation of time: Empirical findings, behavioral models, and problems of measurement. *Journal of Economic literature* **29** (2), 471–522 (1991).

13. Y. Ling, A. Imas, Underreporting of AI use: The role of social desirability bias. *Available at SSRN* (2025).

14. S. Peng, E. Kalliamvakou, P. Cihon, M. Demirer, The impact of ai on developer productivity: Evidence from github copilot. *arXiv preprint arXiv:2302.06590* (2023).

15. M. Hoffmann, S. Boysel, F. Nagle, S. Peng, K. Xu, *Generative AI and the Nature of Work*, Tech. rep., CESifo Working Paper (2024).

16. D. Yeverechyahu, R. Mayya, G. Oestreicher-Singer, The impact of large language models on open-source innovation: Evidence from GitHub Copilot. *arXiv preprint arXiv:2409.08379* (2024).

17. F. Song, A. Agarwal, W. Wen, The impact of generative AI on collaborative open-source software development: Evidence from GitHub Copilot. *arXiv preprint arXiv:2410.02091* (2024).

18. S. Aum, Y. Shin, *Is Software Eating the World?*, Tech. rep., National Bureau of Economic Research (2024).

19. S. Juhász, J. Wachs, J. Kaminski, C. A. Hidalgo, The software complexity of nations. *arXiv preprint arXiv:2407.13880* (2024).

20. K. Handa, *et al.*, Which Economic Tasks are Performed with AI? Evidence from Millions of Claude Conversations. *arXiv preprint arXiv:2503.04761* (2025).

21. R. M. del Rio-Chanona, N. Laurentsyeva, J. Wachs, Large language models reduce public knowledge sharing on online Q&A platforms. *PNAS nexus* **3** (9), pgae400 (2024).

22. E. Andreadis, M. Chatzikonstantinou, E. Kalotychou, C. Louca, C. Makridis, Local Heterogeneity in Artificial Intelligence Jobs Over Time and Space. *Available at SSRN* (2024).

23. D. Bearson, N. L. Wright, Strategic Targeting and Unequal Global Adoption of Artificial Intelligence. *Columbia Business School Research Paper Forthcoming* (2025).

24. R. Janssen, R. Kesler, M. E. Kummer, J. Waldfogel, *GDPR and the lost generation of innovative apps*, Tech. rep., National Bureau of Economic Research (2022).

25. G. Aridor, Y. K. Che, T. Salz, The effect of privacy regulation on the data industry: empirical evidence from GDPR. *RAND Journal of Economics* **54** (4), 695–730 (2023).

26. D. Comin, B. Hobijn, An exploration of technology diffusion. *American Economic Review* **100** (5), 2031–2059 (2010).

27. P. A. David, The dynamo and the computer: an historical perspective on the modern productivity paradox. *The American Economic Review* **80** (2), 355–361 (1990).

28. E. Brynjolfsson, D. Rock, C. Syverson, The productivity J-curve: How intangibles complement general purpose technologies. *American Economic Journal: Macroeconomics* **13** (1), 333–372 (2021).

29. R. Juhász, M. P. Squicciarini, N. Voigtländer, Technology adoption and productivity growth: Evidence from industrialization in France. *Journal of Political Economy* **132** (10), 3215–3259 (2024).

30. E. Paradis, *et al.*, How much does AI impact development speed? An enterprise-based randomized controlled trial, in *Proceedings of the 47th IEEE/ACM International Conference on Software (ICSE)* (IEEE) (2025).

31. P. T. Nguyen, *et al.*, GPTSniffer: A CodeBERT-based Classifier to Detect Source Code Written by ChatGPT. *Journal of Systems and Software* **214**, 112059 (2024), doi:10.1016/j.jss.2024.112059.

32. D. Guo, *et al.*, GraphCodeBERT: Pre-training Code Representations with Data Flow (2021), doi:10.48550/arXiv.2009.08366.

33. T. Ye, *et al.*, Uncovering LLM-Generated Code: A Zero-Shot Synthetic Code Detector via Code Rewriting (2024).

34. H. Fang, J. Herbsleb, B. Vasilescu, Novelty begets popularity, but curbs participation-a macroscopic view of the python open-source ecosystem, in *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering* (2024), pp. 1–11.

35. K. Frenken, M. B. Punt, A new view on radical innovation, in *International Sustainability Transitions conference, Utrecht, the Netherlands* (2023).

36. C. Freeman, C. Perez, Structural crises of adjustment: business cycles. *Technical change and economic theory. Londres: Pinter* (1988).

37. J. Gortmaker, *Open source software policy in industry equilibrium*, Tech. rep., Working paper, Tech. Rep (2024).

38. H. Sackman, W. J. Erikson, E. E. Grant, Exploratory experimental studies comparing online and offline programming performance. *Communications of the ACM* **11** (1), 3–11 (1968).

39. G. E. Bryan, Not all programmers are created equal, in *Proceedings of 1994 IEEE Aerospace Applications Conference Proceedings* (IEEE) (1994), pp. 55–62.

40. L. Betti, L. Gallo, J. Wachs, F. Battiston, The dynamics of leadership and success in software development teams. *Nature Communications* **16** (1), 1–11 (2025).

41. A. Humlum, E. Vestergaard, *The adoption of ChatGPT*, Tech. rep., IZA Discussion Papers (2024).

42. H. Bao, M. Sun, M. Teplitskiy, Where there's a will there's a way: ChatGPT is used more for science in countries where it is prohibited. *Quantitative Science Studies* pp. 1–23 (2025).

43. A. R. Doshi, O. P. Hauser, Generative AI enhances individual creativity but reduces the collective diversity of novel content. *Science Advances* **10** (28), eadn5290 (2024).

44. GitHub, GitHub GraphQL API Documentation (2024), `https://docs.github.com/en/graphql`, accessed: 2024-02-27.

45. D. Spadini, M. Aniche, A. Bacchelli, PyDriller: Python Framework for Mining Software Repositories, in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (ACM, Lake Buena Vista FL USA) (2018), pp. 908–911, doi:10.1145/3236024.3264598.

46. M. Chen, *et al.*, Evaluating Large Language Models Trained on Code (2021), doi:10.48550/arXiv.2107.03374.

47. Q. Zheng, *et al.*, CodeGeeX: A Pre-Trained Model for Code Generation with Multilingual Benchmarking on HumanEval-X, in *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, KDD '23 (Association for Computing Machinery, New York, NY, USA) (2023), pp. 5673–5684, doi:10.1145/3580305.3599790.

48. Z. Feng, *et al.*, CodeBERT: A Pre-Trained Model for Programming and Natural Languages, in *Findings of the Association for Computational Linguistics: EMNLP 2020*, T. Cohn, Y. He, Y. Liu, Eds., pp. 1536–1547, doi:10.18653/v1/2020.findings-emnlp.139.

49. D. H. Autor, D. Dorn, The growth of low-skill service jobs and the polarization of the US labor market. *American Economic Review* **103** (5), 1553–1597 (2013).

Acknowledgments

We thank Hillary Vipond, Andrea Musso, and participants of the ANETI Brownbag seminar for helpful comments. We thank Márton Salamon for valuable research assistance.

Funding: F. N., S.D., J.W. and X.F. received financial support from the Austrian Research Promotion Agency (FFG), project #873927 (ESSENCSE). JW also acknowledges financial support from the Hungarian National Scientific Fund (OTKA FK 145960).

Author contributions: S.D. and F.N. conceptualized the research. S.D. implemented the primary method. S.D. and J.W. collected the data. X.F. collected data and estimated the volume of the U.S. programming-related wage sum. All authors analyzed the data. F.N. and J.W. led in drafting the manuscript. All authors contributed to the writing of the manuscript.

Supplementary materials

Materials and Methods

Supplementary Text

Figs. S1 to S3

Tables S1 to S4

References *(7-49)*

Movie S1

Data S1

Supplementary Materials for

Who is using AI to code?
Global diffusion and impact of generative AI

Simone Daniotti*, Johannes Wachs, Xiangnan Feng, Frank Neffke

This PDF file includes:

Other Supplementary Materials for this manuscript:

Materials and Methods

Data

Our data collection proceeds as follows. First, we geolocate GitHub users using self-reported locations in GitHub profiles obtained through the *GraphQL* API (*44*). Fig. S1 of the SI shows that these self-reported locations closely match the ones derived from users' IP addresses. Next, we collect all commits by the about 300k users that report US locations to projects using Python as their programming language. To do so, we recursively clone all GitHub directories related to a given project. Finally, we obtain all commits to these projects, which we analyzed using the *PyDriller* tool (*45*). To manage computational expenses, the analysis of the other countries in Fig. 2**B** is based on a random sample of 2,000 programmers per country-year, leading to a total of analyzed 70,000 user-year observations).

Detecting generative AI in code

To detect the use of generative AI, we focus on the most elemental code block that represents the solves a specific task and presents also informative text about it: functions. For each function, we determined how many lines were modified in the commit, keeping only those functions in which over 80% of the total lines of code contained modifications.

Training data. The ground truth data on which we train our supervised model for detecting AI generated Python functions are compiled by combining multiple sources and techniques. We start by collecting functions created by human coders from three different datasets. The first dataset randomly samples Python functions from GitHub that were created before the introduction of AI tools to support programmers. In particular, we sample code created in 2018. The second and third datasets are HumanEval (*46*), HumanEval-X (*47*), both of which were originally created to evaluate and measure functional correctness of code. Together, these datasets contain samples of 4,000 human-written Python functions. Adding the latter two datasets ensures that the human-made Python functions we use to train our models include examples created in different years, including years in which generative AI tools had become more widely available. Next, inspired by (*33*), we generate a synthetic dataset of Python functions written by different LLMs, using GPT3.5-turbo, GPT-4, GPT-4o-mini. To do so, we create synthetic clones of the human-written functions described above, using an LLM chain that combines two LLMs, inspired by recent advances of LLM Agents. Each LLM performs a different task. The first LLM is prompted to describe a given human-made function in terms of its functionality and the structure of the required input and generated output. The second LLM is asked to read this description, and to then generate a function that accomplishes the described task. The exact prompts and an example of the output is found in Table S1 of the SI.

Detection model. Our generative AI detection model relies on open source components and is set up to efficiently scale for analyzing millions of Python functions. We chose a state-of-the-art technique based on Codebert (*48*). Our approach resembles that of GPTSniffer (*31*). In this paper, the authors train a CodeBert LLM to detect AI-written Javascript programs. We try to detect AI-written Python functions. To do so, we build a classifier on top of a newer and more advanced version of the LLM, GraphCodeBert (*32*). The latter is better able to capture and understand patterns in code. To the Standard GraphCodeBert, we add a linear layer to perform a classification task. We then finetune all parameters in the training set to optimize model predictions.

Tokenization was performed using GraphCodeBERT's tokenizer, setting a maximum sequence length of 512 and applying padding and truncation to maintain consistency across inputs. As a loss function, we use cross-entropy loss (`torch.nn.CrossEntropyLoss`), as common in RoBERTa-based classification models. This function is well-suited for classification problems. As shown in Fig. S2 of the SI, in our case it learns to effectively differentiate between AI-generated and human-written code. Since the model returns raw logit values, the model internally applies a softmax operation, comparing the predicted probability distribution with the ground truth labels.

To train the model, we split our ground truth data into training and evaluation sets using an 80/20 ratio with a fixed random seed. We then use the Hugging Face Trainer API to handle training, evaluation, and optimization. Our training configuration includes 10 epochs, a batch size of 32 per device for both training and evaluation and the AdamW optimizer (`adamw_hf`) with a learning rate of 1e-5 and weight decay of 0.005. We set `warmup_steps=1000` to help stabilize learning in the early phases. Logging occurs once every 100 steps, with evaluation and model check-pointing at each epoch. In the end, only the best-performing model on the evaluation set is retained. For reproducibility, we use a fixed data seed (`seed 365`). As shown in Fig. S2 of the SI, the model manages to effectively identify AI generated code in our ground truth data, reaching an out-of-sample ROC AUC Score of 0.9643 and Average Precision of 0.9685.

Role	**Prompt**
System	You are a python expert programmer.
User (Generate description)	This is a python script in markdown. Describe the task or tasks this script is solving, explaining the input and the output specifications for each function.
User (Code from description)	This is the description of a python script. Based on the description, write a full code that fulfills that task/tasks. The python script should be organized in a single markdown block. Please return only the code, do not return any clarifications before or after the code.

Table S1: System and User Prompts for Artificial Dataset generation.

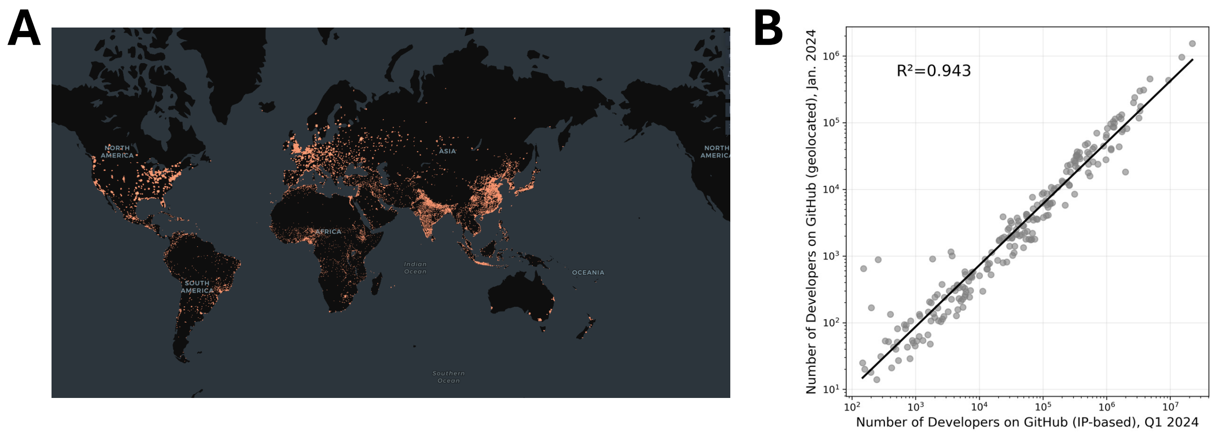
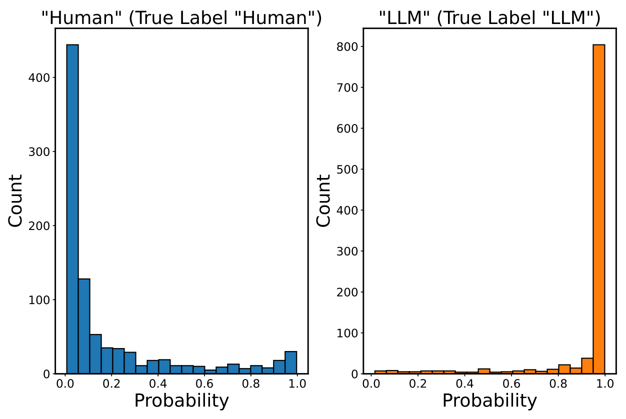


Figure S1: **Location of GitHub users. A**: Self-reported locations of GitHub users. **B**: Number of users in each country based on self-reported locations against IP addresses.

Estimating AI usage rates

In Fig. 2 and in the regression analyses, we study the diffusion and effects of generative AI by quantifying the probability that a piece of code was written by AI: $P(A = 1)$. Our data allow us to estimate the probability that our models detects the use of AI in a certain function: $P(D = 1)$. Using the law of total probability, we can write the latter probability as:

$$P(D = 1) = P(D = 1|A = 1)P(A = 1) + P(D = 1|A = 0)P(A = 0) \quad \text{(S1)}$$



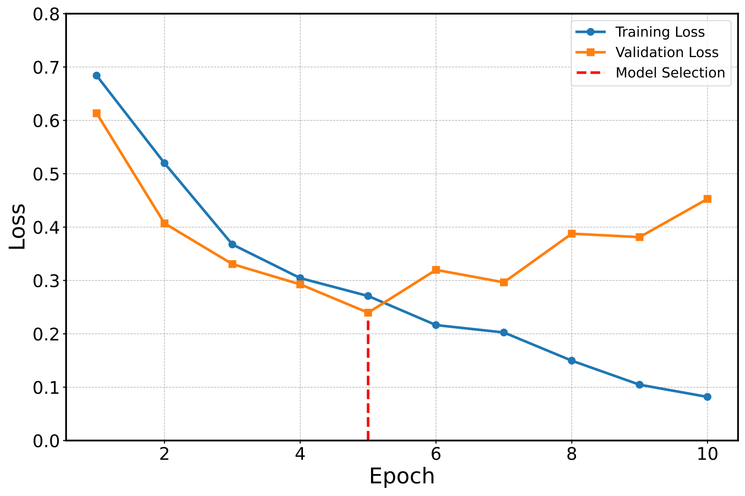


Figure S2: **Detector Prediction Test.** Evaluation of the trained detector on a test set. **A**: predicted probability that code was AI-generated for human-generated functions. **B**: predicted probability that code was AI-generated for only AI-generated functions. **C**: Loss curve AI detection model.

or

$$P(D = 1) = P(D = 1|A = 1)P(A = 1) + P(D = 1|A = 0)\,(1 - P(A = 1))$$

We can estimate some of these terms using our ground truth data set:

- $P(D = 1|A = 1)$: $\hat{d}_{GT}^{AI}$, relative frequency of AI-generated code in ground truth data detected to be AI (true positive rate)
- $P(D = 1|A = 0)$: $\hat{d}_{GT}^{hum}$, relative frequency of human-written code in ground truth data detected to be AI (false positive rate)
- $P(D = 1)$: $\hat{d}$, relative frequency of observed quantity (AI detection rate):

- $P(A = 1)$: $\hat{y}$, estimated quantity of interest (AI usage rate)

Using this notation, we can write eq. (S1):

$$\begin{aligned} \hat{d} &= \hat{d}_{GT}^{AI}\hat{y} + \hat{d}_{GT}^{hum}(1 - \hat{y}) \\ &= \hat{y}\left(\hat{d}_{GT}^{AI} - \hat{d}_{GT}^{hum}\right) + \hat{d}_{GT}^{hum} \end{aligned}$$

and rearrange terms to arrive at:

$$\hat{y} = \frac{\hat{d} - \hat{d}_{GT}^{hum}}{\hat{d}_{GT}^{AI} - \hat{d}_{GT}^{hum}}, \tag{S2}$$

$\hat{y}$ corrects the AI adoption probabilities our model provides for miss-classification errors when averaging such probabilities across functions. We use this quantity throughout the paper as the estimated AI adoption rate in a given sample of functions.

Detailed regression results

To estimate the association between AI adoption rates and demographic variables, as well as the effect of AI adoption on users' output, we first need to determine the adoption rate of a given user at a given point in time. Because we can only detect AI adoption in functions, we create for each individual a dataset of time-stamped functions. Next, we sort these functions by the date of the commit in which they were created or altered. Finally, we calculate a moving average of the AI adoption rates detected by our model, using the correction in eq. (S2). The final result is a time-series that reflects how a user's (latent) adoption rate of generative AI changes over time.

We experiment with different window-widths ,as in Fig. S4. Wider windows average over a larger number of functions and therefore reduce measurement errors. This is evident in an increase of effect sizes as windows become larger, which we interpret as the result of a reduction in the attenuation bias associated with errors-in-variables. However, widening the windows also reduces the time resolution with which we determine adoption rates. In the main text, we report results using eight observed functions per window.

Finally, we need to determine a user's AI adoption rate for commits, including for that do not contain any altered functions. To do so, we link each commit to the window whose average time stamps are closest to the time stamp of the commit.

AI adoption and user demographics. We first estimate how AI adoption rates change with experience. We proxy experience as the number of years since a user's first recorded activity on GitHub. Because our data starts in 2011, this experience is right-censored. The longest experience category therefore contains individuals with the stated number of years of experience or more. To estimate the average AI-adoption rates at different levels of experience, we use our sample of US users. Next, we regress adoption rates in 202X on a set of dummies that encode the user's years of experience. To allow for correlated errors within the same user, we calculate standard errors clustered at the user level. Point estimates with their confidence intervals are plotted in Fig. 3, further details are provided in Table S4.

To estimate usage rates by gender, we first infer a user's gender using Gender-Guesser `https://pypi.org/project/gender-guesser/`, a dictionary-based method to infer a user's gender based on their first name and country (here, the US). As a consequence, this gender may not accurately reflect the gender the user identifies with. We categorize users where the algorithm does not provide a label with high-confidence as unknown. We then proceed with a similar regression analysis, replacing the experience dummies by a gender dummy. Point estimates with their confidence intervals are plotted in Fig. fig:3, further details are provided in Table S5.

AI adoption and output. To determine the effects of AI adoption on the quantity and nature of output that GitHub users produce, we aggregate the data to user-quarter cells, leading to a sample of 100 thousand users that we observe, on average for 4 quarters. Next, we rely on so-called fixed effects models, estimating Ordinary Least Squared models with user and quarter fixed effects. This allows us to estimate the effects of AI by comparing how output changes with changes in AI adoption within the same user, keeping constant user characteristics that do not change over time, as well controlling for secular changes that affect all users. We again focus on the commits by US users. To determine the precision of our estimates, we use two-way clustered standard errors, allowing for correlations in errors within individuals, as well as within quarters.

We estimate these models for two types of variables. The first measures the activity rate of users in terms of the number of commits a user makes in each quarter. The second aims to quantify changes in the nature of code that users produce. To do so, we describe commits in terms of the libraries and combinations of libraries they add. The rationale for this is that different libraries facilitate introducing different types of functionality in a script. The libraries, and especially the combinations of libraries, therefore provide a rough indication of what kind of program a user works on. If we observe that users start using new libraries or library combination, either compared to their own past projects or compared to any project in our dataset, we interpret this as a sign that the user experiments with new types of code. This is in line with prior work that interprets new library combinations as a sign of innovation in code (*34*). To analyze such experimentation, we count the number of libraries or library-combinations that users add in a commit in a given quarter that we had not yet observed before. We initialize this against the set of libraries and library combinations in 2019. We distinguish between two types of experimentation. First, we count the number of libraries or library combinations that are new to the user. Second, we count the number of libraries or library combinations that are new to the world, i.e., that are not observed anywhere in our dataset.

We use both types of variables as dependent variables after log-transforming them. To avoid $\log(0)$ issues, we increment each count by 1 unit before taking logs. We then estimate the following models:

$$\log(N_{i,q}^{\texttt{type}} + 1) = \beta_{AI}^{\texttt{type}} \hat{y}_{i,q} + \eta_i + \tau_q + \varepsilon_{i,q},$$

where $N_{i,q}^{\texttt{type}}$ counts for user i in quarter q the number of events in one of the following categories: commits, libraries new to the user, libraries new to the world, library combinations new to the user, or library combinations new to the world. η_i denotes user fixed effects and τ_q denotes quarter fixed effects. $\hat{y}_{i,q}$ is the average estimated AI usage rate across commits by individual i in quarter 1, as defined in eq. (S2) with a window of size 8.

Parameter estimates $\hat{\beta_{AI}}^{\texttt{type}}$ can be interpreted as semi-elasticities. That is they approximately describe by which percentage $N_{i,q}^{\texttt{type}}$ changes when adoption rates go from 0 to 100%. For instance, $0.3\hat{\beta_{AI}}^{\texttt{commits}}$ the percentage change in commits attributed to the average US adoption rate towards the end of 2024.

	Unique Library Combinations (log +1)	Unique Libraries (log +1)	Unique Library Combinations (new-to-user) (log +1)	Unique Libraries (new-to-user) (log +1)	Unique Library Combinations (new-to-world) (log +1)	New Unique Libraries (new-to-world) (log +1)
Mean AI Share	0.063*** (0.016)	0.111*** (0.019)	0.108*** (0.032)	0.070** (0.027)	0.098*** (0.029)	0.018 (0.014)
Quarter FE	x	x	x	x	x	x
User FE	x	x	x	x	x	x
Observations	287,110	287,110	103,784	103,784	103,784	103,784
S.E. Clustered				by: User		
R^2	0.551	0.490	0.571	0.443	0.589	0.464

Table S2: Effect of AI usage rate on library usage. Stars denote significance levels: *** $p < 0.001$, ** $p < 0.01$, * $p < 0.05$. Standard errors, two-way clustered by user and by quarter in parentheses. To initialize the number of new unique libraries and unique libraries combinations, we use activity the year in 2019 as our baseline. Next, we update quarter-by-quarter the unique entries in users' own commit history, as well as in the universe of entries in our dataset.

One potential concern is that our estimates are confounded by an omitted variable. For instance, if our AI detection model systematically miss-classifies code by users that differ in their coding output, this would confound our estimates. To analyze this, we perform a placebo test: we estimate the effect of user's —detected— AI usage rates before the introduction of generative AI tools. That is, we rerun our analyzes in a sample that only contains quarter before the year in which co-pilot was launched, i.e., before 2022. In this period, the detected AI usage shares do not carry any information on how much users rely on generative AI.

Results are shown in Fig. S3. Unlike our results for the entire period, we now find no evidence that (erroneously) detected AI usage is statistically significantly associated with higher levels of activity or experimentation: p-values of our AI variable range from 0.05 to 0.22.

Finally, we analyze the potential role of attenuation bias—the tendency of measurement error to bias parameter estimates toward zero—in our results. Although our AI detection model performs well, at the user level, we infer AI usage rates from the functions they produce. The greater the number of functions we can analyze, the more precise our measurement of the (latent) AI usage rate will be. Consequently, our estimates of AI usage at the level of entire countries will be relatively precise. Moreover, attenuation biases occur when independent variables are miss-measured. Our analyses where AI usage rates feature as dependent variables are not affected by such biases.

To analyze to what extent attenuation biases may distort our user-level output analyses, we exploit the fact that the estimated user-level AI usage rates are based on moving averages across

	Commits (log +1)
Mean AI Score (Rolling)	0.073***
	(0.014)
Quarter FE	x
User FE	x
Observations	287,110
S.E. Clustered	by: Quarter
R^2	0.644

Table S3: Effect of AI usage rate on commit activity. Stars denote significance levels: *** $p < 0.001$, ** $p < 0.01$, * $p < 0.05$. Standard errors, two-way clustered by user and by quarter in parentheses.

Panel		**Panel**	
Exp 1	-0.012	Exp 8	-0.076***
	(0.014)		(0.014)
Exp 2	-0.025	Exp 9	-0.080***
	(0.014)		(0.014)
Exp 3	-0.023	Exp 10	-0.078***
	(0.014)		(0.014)
Exp 4	-0.034*	Exp 11	-0.087***
	(0.014)		(0.015)
Exp 5	-0.049***	Exp 12	-0.088***
	(0.014)		(0.015)
Exp 6	-0.058***	Exp 13	-0.102***
	(0.014)		(0.014)
Exp 7	-0.062***	Intercept	0.466***
	(0.014)		(0.012)
Observations: 28,707	R^2: 0.008	S.E. Clustered by: `User`	

Table S4: Effect of Experience on AI Share. Standard errors clustered at the user level. Significance levels: *** p <0.001, ** p <0.01, * p <0.05.

	Unique Library Combinations (new-to-user)	Unique Libraries (new-to-world)	Commits (log +1)
	(log +1)	(log +1)	(log +1)
Mean AI Share	0.066	-0.002	-0.108
	(0.045)	(0.019)	(0.086)
Quarter FE	x	x	x
Observations	26,494	26,494	60,881
S.E. Clustered	by: User		by: Quarter
R^2	0.037	0.050	0.211

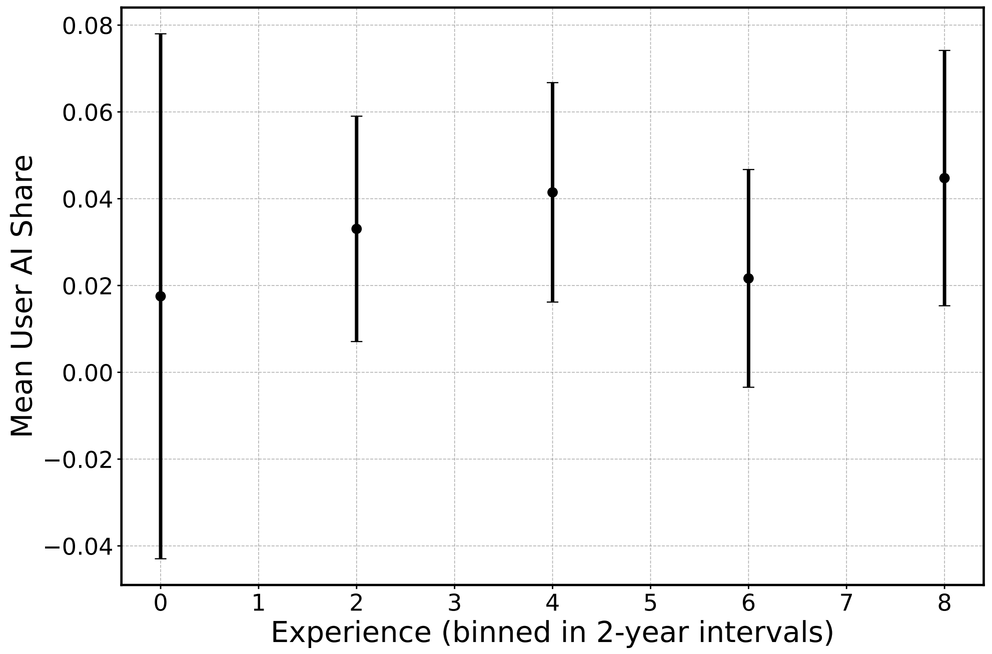


Figure S3: Effect of AI Share on library imports, commit activity, and AI adoption likelihood. The table (left) reports the effect of AI share on library imports and commit activity, using data up to 2021. The plot (right) shows the likelihood of AI adoption by 2019 as a function of user tenure, with 95% confidence intervals.

	AI Share
	(1)
Male	-0.015
	(0.016)
Unknown	-0.004
	(0.015)
Intercept (Female)	0.414***
	(0.015)
Observations	32,011
S.E. Clustered	by: User
R^2	0.000

Table S5: Effect of Gender on AI Share. Standard errors clustered at the user level. Significance levels: *** p <0.001, ** p <0.01, * p <0.05.

multiple functions. By increasing the number of functions used to create such averages, idiosyncratic noise cancels out at a rate of $\frac{1}{\sqrt{n_w}}$, where n_w is the number of functions in the window used for calculating moving averages.

Fig. S4 plots the parameter estimates with 95% confidence intervals of the effect of AI usage rates on three dependent variables: number of commits, number of library combinations that are new to the user and number of library combinations that are new to the world. The estimates use the same model as in Tables S2 and S3. That is, they include user and quarter fixed effects and standard errors are clustered by user and by quarter. Our sample consists of all users for which we have enough functions to create a window of size 20, the maximum window size we consider. Consequently, the analysis is run on the same users, regardless of the window size. Note that this leads to a slightly different sample to the one used in Tables S2 and S3. Parameter estimates for different window sizes are plotted along the horizontal axis. Window size information is expressed in terms of $1/\sqrt{n_w}$, i.e., proportionally to the expected attenuation bias.

Figure S4 show results. For all three dependent variables, we find that effect estimates fall as windows become smaller. The blue markers denote the effects for windows of size 8, the window-size selected in the analysis reported in Fig. 3 of the main text. This window size strikes a balance between substantial noise-reduction and fine-grained enough temporal resolution. Note that enlarging the windows will also reduce the number of users and bias the sample towards more active users that create a larger number of functions. The graphs in Fig. S4 suggest that measurement biases are likely to attenuate estimated effects towards zero. Therefore, the effects reported in the main text are likely to be conservative. That is, improving the precision of our AI detection model may lead to even greater effects of AI usage on activity and experimentation rates.

Estimating the total wage sum related to programming tasks in the US

Estimating how much the US economy spends on programming tasks is not trivial. On the one hand, although the US Bureau of Labor Statistics' (BLS) Standard Occupation Classification (SOC) lists programming-related occupations, such as *Computer programmers* and *Software developers*, these jobs entail more than just coding tasks. On the other hand, a wide range of other occupations may not focus on programming but still substantial programming tasks to be carried out, from *Online merchants* to *Statisticians*. To estimate how much time workers in each of the almost 900 occupations in the SOC classification spend on programming tasks, we rely on data from the Occupational Information Network (O*NET). Next, we combine this estimate with information from the BLS' Occupational Employment and Wage Statistics (OEWS) and the American Community Survey (ACS) on employment and wages in these occupations to arrive at an estimate of the overall wage sum in the US that is associated with programming tasks. We link these data sources at the most detailed, 6-digit, level of the SOC classification.

O*NET is the primary data source for occupational information in the US. It conducts surveys and expert analysis of occupations to determine a variety of characteristics of jobs. Here, we mainly use the information O*NET contains on the tasks that occupations require. We focus on the *Task Ratings* file O*NET 29.2, released in February 2025, which lists around 20 tasks for each occupation, amounting to about 17k distinct tasks. For each task, this file lists how often it is performed and how important it is in the job. The frequency information is encoded in a seven-item variable group that provides information on which percentage of workers in the occupation performs with a given frequency, ranging from 1, "yearly or less," to 7, "hourly or more." For example, 34.85% of *Online Merchants* perform the task *Receive and process payments from customers, using electronic transaction services* "daily" (level 5), and 23.91% *Calculate revenue, sales, and expenses, using financial accounting or spreadsheet software* "more than weekly" (level 4).

To convert these frequencies into estimates of the share of time that workers in a given occupation spend on each task, we explore two approaches. In the first ("distributive") approach, we try to make reasonable assumptions about how much time a worker spends on tasks in a given frequency category, as listed in the third column of Table S6 below. Next, we assume that workers distribute the time allotted to each frequency category and weight this by the percentage of responders of the tasks in the category. For example, according to Table S6 tasks performed "several times daily" (level 6) are assigned a weight of 0.25. That is, we assume that, taken together, tasks with level 6 amount to 25% of the working time in an occupation. If the occupation contains three tasks at frequency level 6 with weights $a\%$, $b\%$, and $c\%$, then these three tasks together, with weights $\frac{a}{a+b+c}$, $\frac{b}{a+b+c}$, and $\frac{c}{a+b+c}$, compose 25% of total working time.

frequency scale	category description	weights (distributive)	weights (relevance)
1	Yearly or less	0	0.5
2	More than yearly	0.02	1
3	More than monthly	0.05	4
4	More than weekly	0.08	48
5	Daily	0.1	240
6	Several times daily	0.25	480
7	Hourly or more	0.50	1920

Table S6: Scales, description, and weights under the two approaches to turn frequency categories in the *Task Ratings* O*NET file into duration shares.

In the second ("relevance") approach, we instead interpret the frequency information for each task as weights that directly express the amount of time workers spend on this task. To do so, we choose a weight for each of the seven frequency categories. These weights are listed in the fourth column of Table S6. Each weight is multiplied by the worker share information. Next, these products are summed and normalized such that they add up to one within each occupation.

Finally, we need to determine how much of the time that is spent on each task is dedicated to programming. To estimate this, we rely on an LLM. For reproducibility purposes, we choose an open source model, Llama 3.3. We provide the model with three different prompts —listed in Section — to arrive at three different estimates of the programming intensity of each task. Table S8 provides examples of tasks and extent to which they require programming using all three prompts.

To arrive at the total wage sum related to programming tasks in the US in 2023, we use two different datasets. The first uses information taken from the BLS on employment and wages:

$$\text{Programming Wage Sum}^{BLS} = \sum_{o} \text{annual wage}_o^{BLS} \times \text{employment}_o^{BLS} \times \tag{S3}$$

$$\sum_{t \in \Theta_o} \text{working time}_{t,o} \times \text{programming share}_{t,o}, \tag{S4}$$

where annual wage_o^{BLS} is the average annual wage in occupation o reported by the BLS, employment_o the number of employees in occupation o according to the BLS, share $\text{time}_{t,o}$ the estimated share of time that workers in occupation o spend on task t based on O*NET information, and programming $\text{share}_{t,j}$ the LLM's estimated share of task t spent on programming.

The second dataset is the 2023 American Community Survey (ACS). The ACS contains a 1% weighted random sample of individuals in the US. We aggregate these data to the occupation level, using the sampling weights as follows:

$$\text{Programming Wage Sum}^{ACS} = \sum_{i} w_i \times \text{annual wage}_i^{ACS} \tag{S5}$$

$$\sum_{t \in \Theta_{o(i)}} \text{working time}_{t,o(i)} \times \text{programming share}_{t,o(i)}. \tag{S6}$$

where $w(i)$ is the frequency weight of individual i in the ACS, annual wage_i^{ACS} the annual salary listed for individual i and $o(i)$ individual i's occupation.

Unlike the BLS data, the ACS samples individuals of all ages and employment statuses (including self-employed and parttime workers). We follow prior literature (*49*) to adjust top-coded annual wages and filter individuals to the active working age population. Occupations in ACS are sometimes slightly more aggregated than in O*NET: about 110 SOC codes in ACS correspond to multiple occupation titles in O*NET. In these cases, we merge the disaggregated occupations of O*NET into the corresponding occupations in the ACS with averaged O*NET scores.

Figures S5 provide scatter plots that compare our different estimates of the amount of time spent on programming tasks in each occupation to the importance of *programming* skills in the occupation's as reported in O*NET. The graph shows that all different approaches yield estimates that correlate strongly with the importance of programming in an occupation's skill requirements, with correlations ranging between .76 and .80. In general, the correlation is highest for the third, most detailed, prompt.

	distributive		relevance	
	BLS	ACS	BLS	ACS
prompt 1	640.63	734.19	647.06	746.10
prompt 2	440.42	528.40	442.70	533.25
prompt 3	468.39	525.93	465.73	527.41

Table S7: Estimated wage sum for programming tasks in the US in 2023 Billions of USD.

Table presents estimated wage sums for programming task in the US based on the two different samples, the three different prompts and the two different approaches to turn frequency in time-share information. Wage sums range from 440B to 746B USD. Note that ACS based estimates always exceed BLS based results. This is because BLS only counts full-time employees and omits self-employed individuals, while the ACS samples individuals regardless of their employment status. Wage sums based on first prompt are generally higher than those of the two prompts, whose estimates are very close to one another. This difference is mainly driven by tasks that are only marginally related to programming, receiving a score of 1 or 2 out of 5. Overall, the wage sums reported in Table imply that $2 \sim 3\%$ of US GDP is spent on remunerating pure programming work.

Prompts adopted to get the programming share of tasks

We supply three different prompts to a Llama 3.3 model to estimate which percentage of the time workers dedicate to a specific task listed in O*NET consists of pure programming work. The first prompt returns a score on a scale that ranges from 1 to 5, where 1 signals that 0% of working time is dedicated to programming and 5 means that almost all working time is used for programming. The detailed prompt is:

```
Analyze the relationship between a specific job task and the skill of computer
    programming.

**Job Role:** ‘‘  ’’

**Task Description:** ‘‘  ’’

**Instructions:**

1.  Consider the definition of "computer programming" as the act of writing, modifying,
    testing, debugging, or maintaining code or scripts (e.g., using languages like Python
    , Java, C++, SQL, shell scripts, PowerShell, etc.).

2.  Evaluate how much of the *specific task described above* involves performing computer
     programming activities. Do not evaluate the entire job role, only the task provided.

3.  Provide *only* a single numerical score from 1 to 5 based on the scale below. Do not
    add any explanation or text other than the score itself.
```

```
**Scoring Scale:**

* **1:** This task is not related to computer programming at all.

* **3:** Performing this task involves spending around half of the time on computer
    programming activities.

* **5:** This task is very related to computer programming, and performing it involves
    spending almost $90\%$ of the time on computer programming activities.

**Score:**
```

The second prompt is a more detailed version of the first one. For each task in each job, it returns a score on a scale that ranges from 0 to 5 where 0 stands for 0% of working time dedicated to programming and 5 almost all time dedicated to programming. The percentage range of each index is given in the prompt. The detailed prompt is:

```
Analyze the relationship between a specific job task and the skill of computer
    programming.

**Job Role:** ‘‘  ’’

**Task Description:** ‘‘  ’’

**Instructions:**

1.  Consider the definition of "computer programming" as the act of writing, modifying,
    testing, debugging, or maintaining code or scripts (e.g., using languages like Python
    , Java, C++, SQL, shell scripts, PowerShell, etc.).

2.  Evaluate how much of the *specific task described above* involves performing computer
     programming activities. Do not evaluate the entire job role, only the task provided.

3.  Provide *only* a single numerical score from 0 to 5 based on the scale below. Do not
    add any explanation or text other than the score itself.

**Scoring Scale (0-5):**

* **0:** **None.** The task involves absolutely no computer programming activities.
*(Estimated programming proportion: $0\%$)*

* **1:** **Minimal / Trace.** Programming is present but extremely limited or incidental,
     a negligible part of the task.
*(Estimated programming proportion: $1\%$ - $10\%$)*

* **2:** **Minor / Some.** Programming is a recognizable but small part of the task,
    clearly secondary to other activities.
*(Estimated programming proportion: $11\%$ - $25\%$)*

* **3:** **Moderate.** Programming constitutes a significant portion, but typically less
    than or roughly equal to other activities within the task.
*(Estimated programming proportion: $26\%$ - $50\%$)*

* **4:** **Substantial / Major.** Programming is a primary activity, taking up a clear
    majority of the effort for the task.
*(Estimated programming proportion: $51\%$ - $75\%$)*

* **5:** **Main.** Programming is the main activity, taking up all or almost all of the
    effort for the task.
*(Estimated programming proportion: $76\%$ - $10\%$)*

**Score:**
```

The third prompt is even more detailed than the first two. For each task in each job, it returns a score on scale that ranges from 0 to 100 that indicate the the working time percentage of programming in this task. Several examples are provided anchor the scale. The detailed prompt is:

```
**Prompt for Llama 3:**
```

```
**Context:**

You are an AI assistant tasked with analyzing job roles and specific tasks within those
    roles to estimate the proportion of time dedicated to programming activities. I will
    provide you with a job title and a description of a single task performed within that
     job.

**Your Goal:**

Based on the provided job title and task description, estimate the approximate percentage
     of time that would be spent **actively writing, testing, debugging, or deploying
    code** for this specific task. Your output should be **only the numerical percentage
    value**. Focus on inferring the nature of the work from the overall description,
    rather than relying solely on specific keywords.

**Input:**

* **Job Title:** ‘‘

* **Task Description:** ‘‘

**Instructions for Your Analysis (Internal Thought Process - Do Not Include in Output):**

1.  **Holistic Task Understanding in Job Context:**

* Read the ‘Task Description‘ thoroughly. Instead of just looking for keywords like "
    develop" or "code," try to understand the overall objective and the types of
    activities implicitly required to achieve it, considering the typical
    responsibilities of the given ‘Job Title‘.

* For instance, a task described as "resolve customer-reported performance bottlenecks in
     the data processing pipeline" implies deep investigation, potentially code profiling
    , optimization, and testing, even if the word "coding" isn’t explicitly used.

2.  **Infer Programming-Related Activities:**

* Based on your holistic understanding, determine what proportion of the task likely
    involves direct engagement with programming activities (e.g., designing algorithms
    that will be coded, writing new code, modifying existing code, scripting, debugging
    complex systems, implementing tests, or managing code deployment).

* Consider the full software development lifecycle if implied by the task.

3.  **Consider Implied Non-Programming Activities:**

* Also, identify parts of the task that, based on its nature, would likely involve
    significant non-programming activities. This could include extensive research before
    any coding can begin, detailed planning and architecting, writing documentation,
    attending meetings for coordination, user interviews, data gathering and analysis (if
     not directly scripting it), or system monitoring and analysis that doesn’t
    immediately lead to code changes.

4.  **Estimate the Percentage based on Inferred Effort:**

* Determine a single numerical percentage representing your best estimate of the time
    spent on direct programming activities for this *specific task*, based on the
    inferred balance of efforts.

**Output Requirement:**

* Return **ONLY the numerical percentage value**. For example, if you estimate $30\%$,
    output only ‘30‘. Do not include the ’$\%$’ symbol or any other explanatory text.

**Examples of Task Analysis (Illustrative - For your understanding of the analysis
    process only, not the output format for the actual task. Note how the reasoning
    infers activities):**

* **Example 1 (Low Programming):**

* **Job Title (Example):** Software Engineer

* **Task Description (Example):** "Collaborate with the product team to define
    specifications for a new user authentication module."

* **Internal Estimation Logic (Example):** The description emphasizes collaboration ("
    collaborate") and definition of requirements ("define specifications"). This strongly
     suggests activities like discussion, documentation, and planning, which are
    primarily pre-coding. *This specific task* is about laying the groundwork. Estimated
    programming time for *this specific task*: $10\%$.

* **Example 2 (Mid-Range Programming):**

* **Job Title (Example):** Data Scientist

* **Task Description (Example):** "Investigate anomalies in sales data and present
    findings to the marketing department."

* **Internal Estimation Logic (Example):** "Investigate anomalies" might involve some
    scripting for data extraction and initial analysis. However, "present findings"
    implies data interpretation, visualization, report preparation, and communication.
    Estimated programming time for *this specific task*: $35\%$.

* **Example 3 (Mid-Range Programming):**

* **Job Title (Example):** DevOps Engineer

* **Task Description (Example):** "Oversee the migration of our primary application
    servers to a new cloud provider, ensuring minimal downtime and performance continuity
    ."

* **Internal Estimation Logic (Example):** "Oversee the migration" involves planning and
    coordination. While automation scripts (programming) will be part of ensuring "
    minimal downtime and performance continuity," a significant portion involves project
    management and validation. Estimated programming time for *this specific task*: $40\%
    $.

* **Example 4 (Low Programming):**

* **Job Title (Example):** UI/UX Designer

* **Task Description (Example):** "Create interactive prototypes for the upcoming mobile
    application redesign based on user feedback and usability testing results."

* **Internal Estimation Logic (Example):** Prototyping here focuses on design tools and
    user experience demonstration, not general-purpose programming, even if some tools
    have coding-like features. Estimated programming time for *this specific task*: $15\%
    $.

* **Example 5 (High Programming):**

* **Job Title (Example):** Full Stack Developer

* **Task Description (Example):** "Refactor the existing monolithic backend service into
    a set of microservices to improve scalability and maintainability."

* **Internal Estimation Logic (Example):** "Refactor... into a set of microservices" is a
     substantial software engineering effort involving deep code analysis, writing
    significant new code, and extensive testing. Estimated programming time for *this
    specific task*: $80\%$.

* **Example 6 (Very High Programming):**
* **Job Title (Example):** Computer Programmers

* **Task Description (Example):** "Perform or direct revision, repair, or expansion of
    existing programs to increase operating efficiency or adapt to new requirements."

* **Internal Estimation Logic (Example):** This task is the core of what a computer
    programmer does. "Revision, repair, or expansion of existing programs" directly
    translates to reading, understanding, modifying, testing, and debugging code.
    Estimated programming time for *this specific task*: $95\%$.

* **Example 7 (Very High Programming):**

* **Job Title (Example):** Computer Programmers

* **Task Description (Example):** "Write, update, and maintain computer programs or
    software packages to handle specific jobs such as tracking inventory, storing or
    retrieving data, or controlling other equipment."

* **Internal Estimation Logic (Example):** "Write, update, and maintain computer programs
    " is unequivocally direct programming work. This involves the full cycle of coding
    for specific functionalities. Estimated programming time for *this specific task*:
    $95\%$.

* **Example 8 (Very High Programming):**
* **Job Title (Example):** Web Developers

* **Task Description (Example):** "Write supporting code for Web applications or Web
    sites."

* **Internal Estimation Logic (Example):** "Write supporting code" is a direct statement
    of programming activity within the context of web development (e.g., backend logic,
    frontend scripting, API integration). Estimated programming time for *this specific
    task*: $90\%$.

* **Example 9 (Very High Programming):**

* **Job Title (Example):** Bioinformatics Technicians

* **Task Description (Example):** "Write computer programs or scripts to be used in
    querying databases."

* **Internal Estimation Logic (Example):** "Write computer programs or scripts" for
    database querying is a clear programming task, essential for data retrieval and
    analysis in bioinformatics. Estimated programming time for *this specific task*: $90
    \%$.

* **Example 10 (Very High Programming):**

* **Job Title (Example):** Atmospheric and Space Scientists

* **Task Description (Example):** "Develop computer programs to collect meteorological
    data or to present meteorological information."

* **Internal Estimation Logic (Example):** "Develop computer programs" for data
    collection or presentation directly points to software development, likely involving
    data processing, numerical modeling, or visualization coding. Estimated programming
    time for *this specific task*: $85\%$ (allowing for some potential non-coding
    research or data interpretation elements within the broader task).

* **Example 11 (Very Low Programming):**

* **Job Title (Example):** Chief Executives

* **Task Description (Example):** "Appoint department heads or managers and assign or
    delegate responsibilities to them."

* **Internal Estimation Logic (Example):** This task is purely managerial and strategic,
    involving decision-making, leadership, and organizational structuring. There is no
    implied programming. Estimated programming time for *this specific task*: $0\%$.

* **Example 12 (Very Low Programming):**

* **Job Title (Example):** Education and Childcare Administrators, Preschool and Daycare

* **Task Description (Example):** "Teach classes or courses or provide direct care to
    children."

* **Internal Estimation Logic (Example):** This task involves direct pedagogical
    activities, caregiving, and interpersonal interaction, with no programming component.
     Estimated programming time for *this specific task*: $0\%$.

* **Example 13 (Very Low Programming):**

* **Job Title (Example):** Food Service Managers

* **Task Description (Example):** "Test cooked food by tasting and smelling it to ensure
    palatability and flavor conformity."

* **Internal Estimation Logic (Example):** This task involves sensory evaluation and
    quality control related to food, entirely non-programming. Estimated programming time
     for *this specific task*: $0\%$.

* **Example 14 (Very Low Programming):**

* **Job Title (Example):** Gambling Managers

* **Task Description (Example):** "Notify board attendants of table vacancies so that
    waiting patrons can play."

* **Internal Estimation Logic (Example):** This task is operational and communicative,
    focusing on managing customer flow and staff coordination within a gambling
    establishment. No programming is involved. Estimated programming time for *this
    specific task*: $0\%$.
```

```
* **Example 15 (Very Low Programming):**

* **Job Title (Example):** Postmasters and Mail Superintendents

* **Task Description (Example):** "Select and train postmasters and managers of associate
     postal units."

* **Internal Estimation Logic (Example):** This task is focused on human resources,
    management, and training, with no direct programming activities. Estimated
    programming time for *this specific task*: $0\%$.

**Now, please analyze the following and provide ONLY the numerical percentage as output,
    focusing on an inference from the overall description:**

* **Job Title:** ‘‘

* **Task Description:** ‘‘

**Begin Analysis and Provide Only the Numerical Percentage. Please only give me the
    numerical percentage and do not output any other text.**
```

Table S8 provides examples of tasks and their scores from the three prompts.

Table S8: Fifteen example tasks from O*NET with their programming share score and corresponding percentage provided by Llama 3.3 using three different prompts.

Task	Prompt 1	Prompt 2	Prompt 3
Write algorithms or programming code for ad hoc robotic applications.	5 (87.5%)	5 (88%)	95 (95%)
Write program code to analyze data with statistical analysis software.	5 (87.5%)	5 (88%)	90 (90%)
Develop efficient and effective system controllers.	5 (87.5%)	5 (88%)	80 (80%)
Analyze problems to develop solutions involving computer hardware and software.	4 (62.5%)	4 (63%)	80 (80%)
Devise or apply independent models or tools to help verify results of analytical systems.	4 (62.5%)	4 (63%)	70 (70%)
Develop or recommend network security measures, such as firewalls, network security audits, or automated security probes.	4 (62.5%)	3 (38%)	60 (60%)
Design, build, or operate equipment configuration prototypes, including network hardware, software, servers, or server operation systems.	3 (37.5%)	3 (38%)	60 (60%)

Implement system renovation projects in collaboration with technical staff, engineering consultants, installers, and vendors.	2 (12.5%)	2 (18%)	40 (40%)
Develop or document reverse logistics management processes to ensure maximal efficiency of product recycling, reuse, or final disposal.	2 (12.5%)	2 (18%)	20 (20%)
Reconcile records of bank transactions.	2 (12.5%)	1 (5.5%)	10 (10%)
Record operating data such as products and quantities pumped, stocks used, gauging results, and operating times.	2 (12.5%)	1 (5.5%)	5 (5%)
Locate underground services, such as pipes or wires, prior to beginning work.	2 (12.5%)	1 (5.5%)	5 (5%)
Set goals and deadlines for the department.	1 (0%)	0 (0%)	0 (0%)
Spread roofing paper on surface of foundation, and spread concrete onto roofing paper with trowel to form terrazzo base.	1 (0%)	0 (0%)	0 (0%)
Treat animal illnesses or injuries, following experience or instructions of veterinarians.	1 (0%)	0 (0%)	0 (0%)

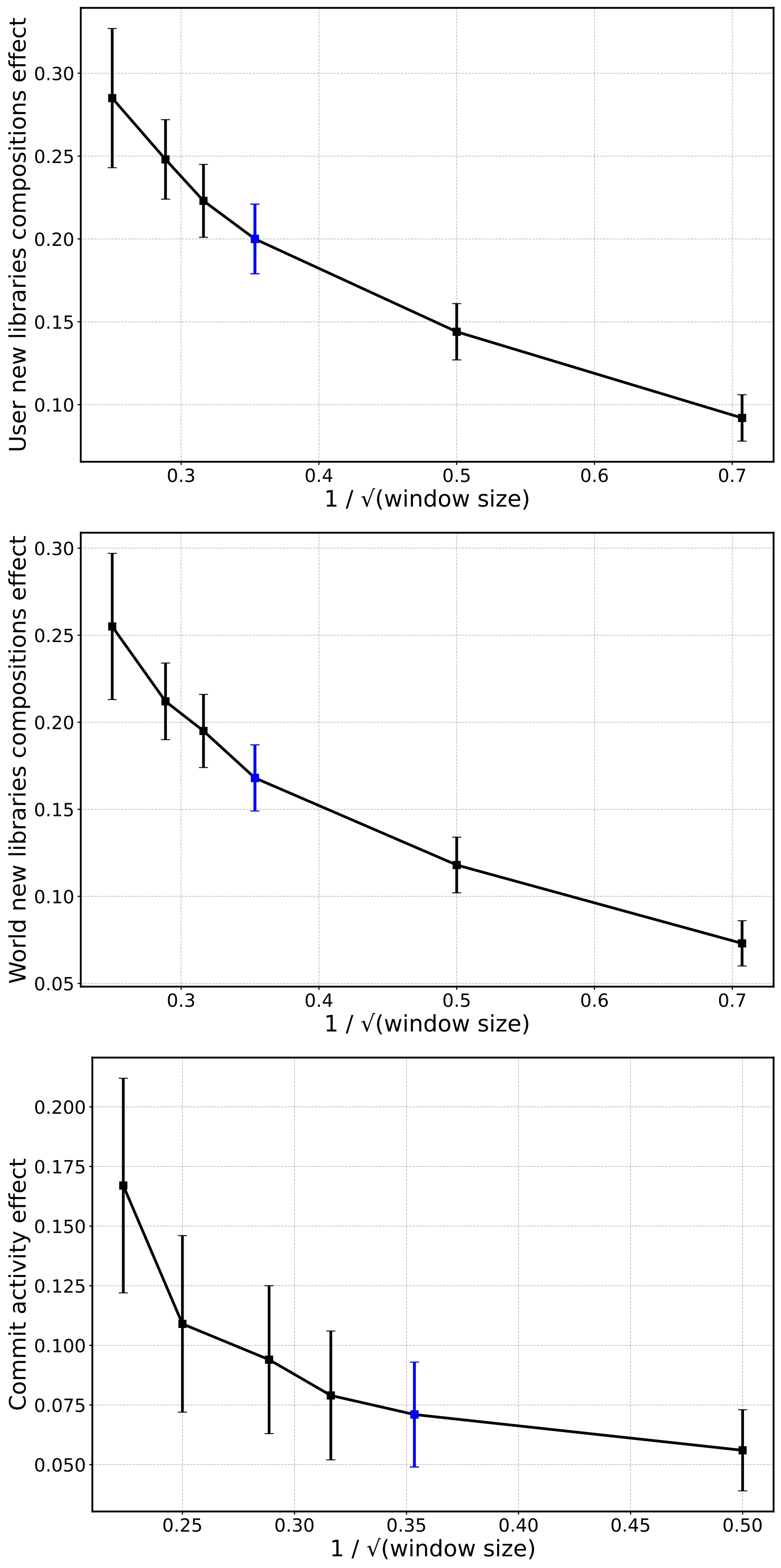


Figure S4: Estimates against AI aggregation window. Each panel plots the estimated effect using a different window size for aggregating AI use: (top) new-to-user, (center) new-to-world, and (bottom) commits. Increasing the window size reduces estimation noise and smoothens the trends.

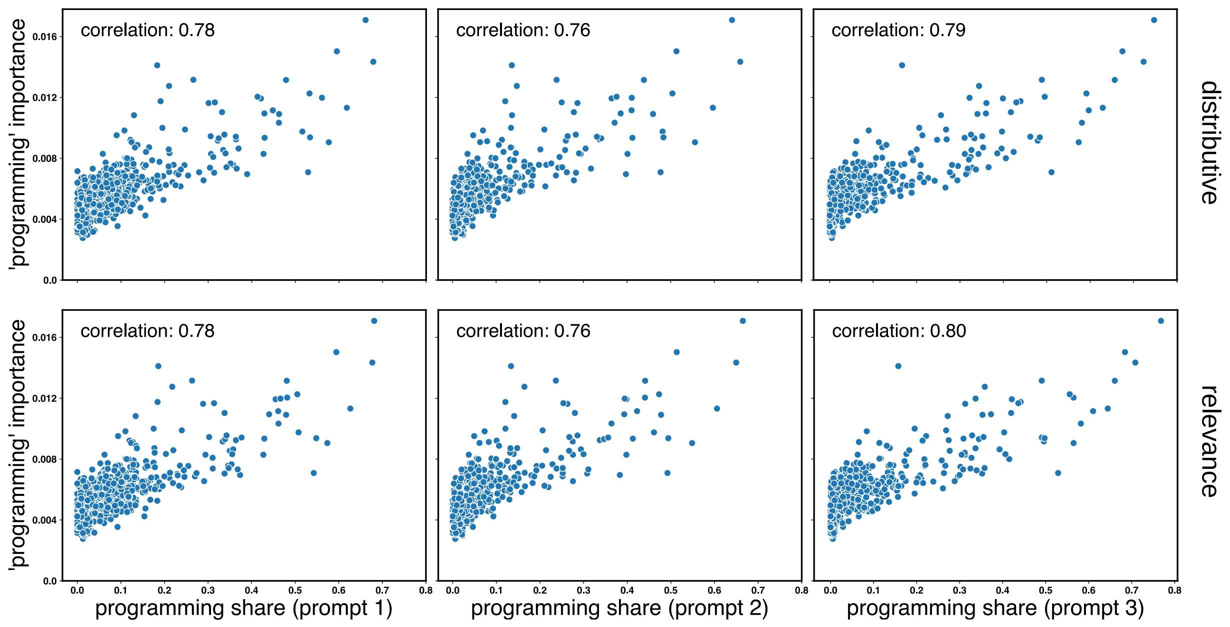


Figure S5: Programming shares. Scatter plots of programming shares by occupation based on the three different prompts and the distributive (top row) or the relevance (bottom row) approach of turning frequencies into time shares against the importance of *programming* skills as reported in O*NET