

Tab 1

RUN LABEL: 1

Prompt: A

Model: ChatGPT 5.2 Auto

Paper: The Impact of AI on Developer Productivity: Evidence from GitHub Copilot

Output:

None

Contribution

The paper provides the first controlled experimental evidence on the productivity impact of a generative AI tool (GitHub Copilot) in professional software development, showing substantial reductions in task completion time and identifying heterogeneous benefits across developer characteristics.

Method

A randomized controlled experiment was conducted in which professional developers were assigned to either a treatment group with access to GitHub Copilot or a control group without it. Participants completed a standardized JavaScript task (implementing an HTTP server). Productivity was measured using task completion time and task success rate, with additional analysis of heterogeneous treatment effects and post-task survey responses.

Data

The study recruited 95 professional developers via Upwork, with 35 completing the task and surveys. Detailed telemetry data from GitHub Classroom captured timestamps, test results, and completion times. Survey data provided demographic information, self-reported productivity estimates, and willingness-to-pay measures.

Findings

Developers using GitHub Copilot completed the task 55.8% faster on average than those in the control group, a statistically significant result. The treated group also showed a slightly higher (though not statistically significant) task success rate. Productivity gains were larger for less experienced developers, those with heavier daily coding workloads, and older developers. Participants underestimated Copilot's actual productivity gains and treated participants reported a higher willingness to pay for the tool.

Limitations

The study focuses on a single, short, standardized programming task in one language, limiting generalizability to other tasks, languages, or real-world collaborative settings. It does not evaluate code quality, security, or long-term effects on developer behavior and skills. The sample size is modest

and drawn from a specific labor market segment (freelancers on Upwork), which may limit external validity.

RUN LABEL: 2

Prompt: B

Model: ChatGPT 5.2 Auto

Paper: The Impact of AI on Developer Productivity: Evidence from GitHub Copilot

Output:

None

Contribution

Presents evidence from a controlled experiment measuring the productivity impact of an AI pair programmer (GitHub Copilot) on professional software development tasks, reporting a substantial reduction in task completion time.

Method

A randomized controlled experiment conducted between May 15 and June 20, 2022. Professional developers recruited via Upwork were randomly assigned to a treatment group with access to GitHub Copilot or a control group without it. Participants completed a standardized JavaScript task (implementing an HTTP server) using GitHub Classroom, with performance measured by task completion time and task success. Surveys were administered before and after the task.

Data

Ninety-five professional developers accepted participation offers; 45 were assigned to the treatment group and 50 to the control group. Thirty-five participants completed the task and survey. Data include timestamps from GitHub Classroom, test suite results (12 checks), demographic survey responses, exit survey responses, and self-reported estimates of productivity gains and willingness to pay.

Findings

Conditioning on task completion, the treatment group completed the task 55.8% faster than the control group (average 71.17 minutes vs. 160.89 minutes), with a statistically significant difference. The treated group's task success rate was 7 percentage points higher, though not statistically significant. Heterogeneous effects indicate larger benefits for less experienced developers, developers with heavier coding workloads, and developers aged 25–44. Treated participants reported higher willingness to pay for GitHub Copilot than control participants.

Limitations

The study focuses on a single standardized programming task and does not examine code quality. Results may not generalize to other tasks, programming languages, or collaborative real-world development settings. The broader economic and labor market implications are not directly measured.

RUN LABEL: 3

Prompt: A

Model: Gemini 3 Fast

Paper: The Impact of AI on Developer Productivity: Evidence from GitHub Copilot

Output:

None

Summary: The Impact of AI on Developer Productivity

-

Contribution

This paper provides the **first controlled experiment** measuring the productivity impact of generative AI tools (specifically GitHub Copilot) in professional software development. It establishes a quantitative benchmark for how AI pair programmers can reduce task completion time and identifies which developer demographics benefit most from these tools.

-

Method

The researchers conducted a **randomized controlled trial** with the following structure:

Task: Participants were required to implement an HTTP server in JavaScript as quickly as possible.

Setup: The experiment used **GitHub Classroom** to track progress. A private repository was created for each dev with a skeleton codebase and a **test suite of 12 checks**.

Groups: * **Treatment Group:** Instructed to use GitHub Copilot (with a 1-minute introductory video).

Control Group: Completed the task without AI assistance but with access to traditional resources like Stack Overflow.

Metrics: 1. **Task Success:** Percentage of participants passing all 12 tests.
2. **Completion Time:** The time elapsed between repository creation and the first commit to pass all tests.

-

Data

Participants: 95 professional freelance developers recruited via Upwork.

Demographics: Primarily aged 25–34, located in India and Pakistan, with an average of 6 years of coding experience and a high education level (4-year degree or higher).

Final Sample: 70 developers (35 from each group) successfully completed both the task and the exit survey.

-

Findings

Massive Speed Increase: The treatment group completed the task **55.8% faster** than the control group (average of 71.17 minutes vs. 160.89 minutes).

Success Rates: The AI group had a slightly higher success rate (7 percentage points), though this was not statistically significant.

Heterogeneous Effects: The benefits of Copilot were most pronounced for **less experienced developers**, older developers (aged 25–44), and those who code for more hours per day.

Perception vs. Reality: Participants in both groups underestimated the AI's impact, predicting only a 35% productivity gain.

Willingness to Pay: The treatment group was willing to pay significantly more for the tool (\$27.25/mo) compared to the control group (\$16.91/mo).

-

Limitations

Task Complexity: The study used a standardized, relatively simple programming task rather than a complex, large-scale collaborative project.

Code Quality: The researchers did not evaluate the **quality, security, or performance** of the code produced, only whether it passed the functional test suite.

Generalizability: The results may vary across different programming languages and specialized technical domains.

Labor Market Impact: The study does not address how these productivity gains would be distributed economically or how they might change the nature of software engineering roles over time.

-

Would you like me to find similar studies comparing other AI coding assistants like ChatGPT or Claude to these GitHub Copilot benchmarks?

RUN LABEL: 4

Prompt: B

Model: Gemini 3 Fast

Paper: The Impact of AI on Developer Productivity: Evidence from GitHub Copilot

Output:

None

Contribution

The paper presents the results of the first controlled experiment to measure the productivity impact of AI pair programmers (specifically GitHub Copilot) on professional software development. It provides empirical evidence that generative AI tools can significantly reduce the time required to complete a standardized programming task.

-

Method

Researchers conducted a randomized controlled experiment involving professional developers. Participants were split into a treatment group (using GitHub Copilot) and a control group (no AI assistance). Both groups were tasked with writing an HTTP server in JavaScript using a skeleton codebase and a test suite in GitHub Classroom. Productivity was quantified by measuring task success and the time elapsed between repository creation and passing all 12 functional tests.

-

Data

The study involved 95 professional programmers recruited from the freelancing platform Upwork. Data points included:

Demographics: Age, location (primarily India and Pakistan), education (mostly 4-year degrees), and income.

Performance Metrics: Task completion timestamps and test suite results.

Survey Responses: Self-reported coding experience, perceived productivity gains, and willingness to pay for the tool.

Findings

Speed: The treatment group completed the task 55.8% faster than the control group (71.17 minutes vs. 160.89 minutes).

Success Rate: The treatment group had a 7 percentage point higher success rate, though this was not statistically significant.

Heterogeneity: Less experienced developers, older developers (25–44), and those with high daily coding loads benefited more from the AI tool.

Perception vs. Reality: Participants underestimated their productivity gains, self-reporting a 35% increase compared to the observed 55.8%.

Willingness to Pay: The treatment group was willing to pay significantly more for the tool (\$27.25/month) than the control group (\$16.91/month).

Limitations

Task Scope: The study focused on a single, standardized task rather than long-term collaboration on large, complex, or proprietary projects.

Code Quality: The experiment did not examine the effects of AI on code quality, security, or performance.

Generalizability: Productivity benefits may vary across different programming languages and types of tasks.

External Factors: The study did not explore how AI-powered tools would change job tasks or labor market distributions at a population level.

Would you like me to create a table comparing the performance metrics of the treatment and control groups?

RUN LABEL: 5

Prompt: A

Model: ChatGPT 5.2 Auto

Paper: How AI Impacts Skill Formation

Output:

None

Contribution

This paper investigates how AI assistance affects skill formation, not just productivity, when developers learn new technical concepts. It provides experimental evidence that while AI can help complete tasks, it can impair the development of conceptual understanding, debugging, and code comprehension if used in cognitively passive ways. The paper also identifies distinct patterns of AI use, showing that some forms of AI interaction preserve learning while others undermine it.

-

Method

The authors ran a between-subjects randomized controlled experiment in which participants learned a new Python asynchronous programming library (Trio). Participants were assigned either to:

a control group (no AI assistance), or

a treatment group (access to a GPT-4o-based AI coding assistant).

Participants completed two coding tasks under time constraints, followed by a rigorously designed quiz evaluating conceptual understanding, code reading, and debugging skills. Keystrokes, AI chat logs, task completion times, and surveys were collected to analyze performance and learning outcomes.

-

Data

Participants: 52 professional or freelance Python developers (26 control, 26 AI), all with prior Python experience but no Trio experience

Tasks: Two Trio-based coding tasks (10–20 minutes each in pilot testing)

Evaluation: A 14-question, 27-point quiz covering 7 core Trio concepts

Additional data: AI interaction transcripts, keystroke logs, self-reported learning and task difficulty surveys

-

Findings

Skill formation decreased with AI use: Participants using AI scored significantly lower on the post-task quiz ($\approx 17\%$ lower), with large effect sizes.

No significant productivity gains overall: AI assistance did not significantly reduce task completion time on average.

Greatest skill loss in debugging: The largest performance gap appeared in debugging questions, a critical skill for supervising AI-generated code.

AI usage patterns matter: Participants who used AI for explanations and conceptual clarification retained learning, while those who delegated full code generation learned far less.

Cognitive engagement is key: AI-enhanced productivity is not a substitute for learning; passive reliance harms long-term competence.

-

Limitations

Short-term measurement: Skill formation was assessed over a single session rather than longitudinally.

Single task domain: Results are based on learning one Python library via a chat-based AI interface.

Limited realism: Participants lacked real workplace incentives to deeply master the library.

No human-assistance baseline: The study does not compare AI help to mentorship or peer programming.

Prompting skills not directly measured: Differences in prompting ability may influence outcomes.

RUN LABEL: 6

Prompt: B

Model: ChatGPT 5.2 Auto

Paper: How AI Impacts Skill Formation

Output:

None

Contribution

The paper investigates how AI assistance affects skill formation when developers learn a new programming library, focusing on whether productivity gains from AI come at the cost of conceptual understanding and other core software engineering skills.

Method

The authors conducted a between-subjects randomized controlled experiment in which participants completed two coding tasks using a previously unfamiliar Python asynchronous programming library (Trio), either with access to an AI coding assistant or without it. Skill formation was evaluated using a post-task quiz measuring conceptual understanding, code reading, and debugging abilities.

Data

Data were collected from 52 participants (26 with AI assistance and 26 without), all of whom had prior Python experience but no prior experience with the Trio library. The study used task completion times, quiz scores, AI interaction transcripts, keystroke logs, and pre- and post-task survey responses.

Findings

AI assistance did not lead to significant average improvements in task completion time, but it significantly reduced skill formation as measured by quiz performance. Participants using AI scored lower overall, with the largest deficits in debugging and conceptual understanding. Some participants who fully delegated coding to AI achieved faster completion times, but at the cost of learning. Distinct AI interaction patterns were identified, with cognitively engaged uses of AI preserving learning outcomes.

Limitations

The study focuses on a short, one-hour learning task involving a single programming library and a chat-based AI interface. It does not measure long-term skill development, does not include agentic AI tools or human assistance as comparison conditions, and relies on a quiz-based evaluation rather than real-world performance outcomes.

RUN LABEL: 7

Prompt: A

Model: Gemini 3 Fast

Paper: How AI Impacts Skill Formation

Output:

None

This academic paper investigates the "cost of convenience" when using AI tools for professional development, specifically focusing on how AI assistance influences the acquisition of new technical skills.

Summary of "How AI Impacts Skill Formation"

-

Contribution

The paper provides empirical evidence that while AI assistance can facilitate task completion, it often **impairs the formation of core technical skills** (conceptual understanding, code reading, and debugging). It moves beyond measuring simple productivity to analyze the **process of human-AI collaboration**, identifying specific interaction patterns that either hinder or preserve learning.

Method

The researchers conducted a **between-subjects randomized controlled experiment** with 52 professional/freelance Python developers.

Participants: Divided into a **Treatment group** (access to GPT-4o via a chat interface) and a **Control group** (access to only web search and documentation).

The Task: Participants had to learn and implement the **Trio library** (a Python library for asynchronous programming) to complete two coding tasks.

Evaluation: Mastery was measured via a 14-question post-task quiz covering three domains: **Debugging**, **Code Reading**, and **Conceptual Understanding**.

Data

Quantitative: Task completion times, quiz scores (graded out of 27 points), and Python familiarity (calibrated via a warm-up task).

Qualitative/Behavioral: Keystroke logs and full transcripts of AI chat interactions to categorize user behavior into six distinct patterns.

Self-Reported: Participant surveys on task difficulty, enjoyment, and perceived learning.

Findings

Learning Impairment: The AI group scored significantly lower on the mastery quiz (**17% lower average**) than the control group, with the largest gap found in **debugging skills**.

Inconsistent Productivity: AI did not provide a statistically significant speed boost on average. While a subset of "delegators" finished faster, others spent significant time refining prompts or reading explanations.

Cognitive Engagement Matters: Skill formation depended on *how* the AI was used.

Low Scorers: Delegated entire blocks of code without asking for explanations.

High Scorers: Used AI to ask conceptual questions or requested line-by-line breakdowns of generated code.

The "Safety" Gap: The erosion of debugging skills suggests that developers who rely on AI to learn may eventually lack the expertise required to supervise or verify AI-generated code in high-stakes environments.

Limitations

Short Duration: The study measured "micro-learning" over a 75-minute window rather than longitudinal skill development over months.

Interface Constraints: The study used a chat-based interface; the researchers note that "agentic" tools (which require even less human input) might cause even greater skill erosion.

Incentive Alignment: As a lab study with paid crowd-workers, participants may have lacked the long-term professional incentive to master the library compared to a full-time employee.

Lack of Human Comparison: The study did not compare AI assistance against human mentorship or pair programming.

RUN LABEL: 8

Prompt: B

Model: Gemini 3 Fast

Paper: How AI Impacts Skill Formation

Output:

None

Contribution

The paper investigates the impact of AI assistance on **skill formation** and **knowledge acquisition** in software engineering. It specifically addresses how the use of AI tools during tasks requiring new concepts affects a developer's subsequent ability to understand, read, and debug code independently.

-

Method

The researchers conducted a **between-subjects randomized experiment** involving 52 participants (26 in a control group, 26 in a treatment group).

Task: Participants used the Python **Trio library** (specifically chosen for its unfamiliarity and "structured concurrency" concepts) to complete two coding tasks.

Conditions: The treatment group had access to a **GPT-4o-based AI assistant**, while the control group used only standard web searches and instructions.

Evaluation: Mastery was measured via a post-task **14-question quiz** (27 points total) covering conceptual understanding, code reading, and debugging.

Data Collection: Researchers collected keystroke logs, AI interaction transcripts, and pre/post-task surveys.

-

Data

Participants: 52 professional or freelance programmers with 1+ years of Python experience but no prior Trio library experience.

Quantitative Metrics: Task completion times, quiz scores (standardized via item response theory), and self-reported learning/difficulty ratings on a 7-point scale.

Qualitative Metrics: Analysis of six distinct AI interaction patterns ranging from full delegation to conceptual questioning.

-

Findings

Skill Impairment: AI use significantly reduced skill formation, resulting in a **17% lower average quiz score** (Cohen's $d=0.738$) compared to the control group.

Specific Deficits: The largest performance gap occurred in **debugging questions**, as the control group gained more experience troubleshooting errors manually.

Productivity Paradox: AI assistance did **not provide significant average efficiency gains** in completion time, though it did ensure all treatment participants finished the task (unlike the control group).

Interaction Matters: High quiz scores were maintained by participants who used AI for **conceptual explanations** or cognitively engaged with the generated code. Low scores were tied to "full delegation" patterns.

Perception: AI users found the tasks easier but reported lower self-rated learning than the control group.

-

Limitations

Task Selection: The study used a single library and a chat-based interface; it did not test more automated "agentic" AI tools.

Task Length: Skill formation was measured over a **one-hour window** rather than a longitudinal period (months or years).

Participant Incentives: Being a paid study with crowd-workers, participants lacked the long-term incentive to master the library required in a real job setting.

Evaluation Scope: The study relied on a quiz rather than a secondary coding task or "human-assistance" counterfactual for evaluation.

Prompting Measurement: The study relied on self-reported AI familiarity rather than an objective measure of prompting fluency.

RUN LABEL: 9

Prompt: A

Model: ChatGPT 5.2 Auto

Paper: The Good and Bad of AI Tools in Novice Programming Education

Output:

None

Here are **five key claims** about the impact of AI tools on software engineering (specifically novice programming education) extracted directly from the paper, each with a supporting snippet and a clear citation.

-

Key Claims from *The Good and Bad of AI Tools in Novice Programming Education*

#

Claim

Supporting Quote / Snippet

Citation

1

AI tool familiarity among novice programmers increases dramatically over time.

"The findings revealed a significant increase in AI tool familiarity (from 28% to 100%) and usage among students."

Zviel-Girshin, 2024

2

Student satisfaction with AI tools improves as they gain experience using them.

"Students' satisfaction with AI tools improved over time."

Zviel-Girshin, 2024

3

Novice programmers primarily use AI tools for low-level support tasks rather than full code generation.

"The most prevalent tasks for which novice programmers used AI tools included creating comments (91.7%), identifying and correcting bugs (80.2%), and seeking information (68.5%)."

Zviel-Girshin, 2024

4

AI tools provide meaningful learning benefits, including improved motivation and real-world relevance.

"AI tools significantly enhanced participants' learning experiences by assisting with information retrieval, bug identification, and writing comments... fostering essential literacies and skills that are crucial for understanding technology and preparing them for the future."

Zviel-Girshin, 2024

5

Over-reliance on AI tools can harm conceptual understanding and independent problem-solving skills.

"The risk of students developing an excessive or even blind reliance on these tools... can lead to a superficial understanding of programming concepts and a diminished ability to solve problems independently."

Zviel-Girshin, 2024

RUN LABEL: 10

Prompt: B

Model: ChatGPT 5.2 Auto

Paper: The Good and Bad of AI Tools in Novice Programming Education

Output:

None

Claim

Evidence Snippet

Citation (source_id, chunk_id)

Novice programmers' familiarity with AI tools increased substantially over the course.

"Initially, only 28% of the teams reported feeling familiar with the usage of AI tools. However, by the end of the course, this familiarity had increased to 100%."

(GoodBadAIProgrammingEd, Results-3.1)

Students' usage of AI tools increased over time, even when not required.

"Cohran's Q = 34.839, p < 0.001, meaning that there were significant differences... indicating that students used more AI tools."

(GoodBadAIProgrammingEd, Results-3.2)

Students' satisfaction with AI tool results improved over time.

"The differences in medians were significant... χ^2 (2) = 11.594, p = 0.003, meaning that the answers were better."

(GoodBadAIProgrammingEd, Results-3.3)

The most common AI-assisted tasks for novice programmers were commenting, bug fixing, and information seeking.

"Creation of comments... 91.7%, bugs identification and correction-80.2%, information seeking-68.5%."

(GoodBadAIProgrammingEd, Results-3.4)

Students expressed concerns about over-reliance on AI tools and inaccurate outputs.

"Common concern categories... inaccurate or misleading information... 89.0%, over-reliance on AI tools... 75.3%."

(GoodBadAIProgrammingEd, Results-3.5)

RUN LABEL: 11

Prompt: A

Model: Gemini 3 Fast

Paper: The Good and Bad of AI Tools in Novice Programming Education

Output:

None

Based on the complete text of the paper, here are five key claims regarding the impact of AI tools on software engineering and programming education:

Claim

Supporting Quote

Citation

AI tools improve with user experience through better prompt engineering.

"The differences in medians were significant... meaning that the answers were better. However, since students used the same language model, this result suggests that they improved their prompt-engineering skills over time."

Zviel-Girshin (2024), Section 3.3

AI tools are highly effective for "drudge work" like documentation.

"The most prevalent tasks for which novice programmers used AI tools included creating comments (91.7%)... The efficiency of AI tools also led to the over-automation of some programming tasks, such as generating comments, with 92% of teams fully outsourcing this task to AI."

Zviel-Girshin (2024), Sections 3.4 & 4.3

AI provides immediate feedback that increases developer motivation.

"Additionally, some participants noted that receiving feedback from AI tools when they were stuck boosted their self-efficacy and motivation to learn."

Zviel-Girshin (2024), Section 4.2

AI-generated content often contains "believable" but incorrect logic.

"Common concern categories... included inaccurate or misleading information/answers... [and] incorrect explanations which appear believable to novices."

Zviel-Girshin (2024), Section 3.5

Excessive AI use can turn developers into passive recipients rather than active problem solvers.

"AI is often doing most of the work. This dynamic can erode the learning experience, as students become passive recipients rather than active participants in the problem-solving process."

Zviel-Girshin (2024), Section 4.3

RUN LABEL: 12

Prompt: B

Model: Gemini 3 Fast

Paper: The Good and Bad of AI Tools in Novice Programming Education

Output:

None

Claim

Evidence Snippet

Citation (source_id, chunk_id)

The most frequent task for which novice students used AI was generating code comments.

"Categories that emerged from the analysis... creation of comments mentioned by 67 teams (91.7%)"

(The Good and Bad of AI Tools in Novice Programming Education, 3.4)

Students' satisfaction with AI tool results improved significantly over the 12-week course.

"The Friedman test... comparing the responses to the item, 'I was happy with the results provided by AI tools'... The differences in medians were significant... meaning that the answers were better."

(The Good and Bad of AI Tools in Novice Programming Education, 3.3)

The primary concern among students regarding AI tools is the risk of receiving inaccurate information.

"Common concern categories... inaccurate or misleading information/answers mentioned by 65 teams (89.0%)"

(The Good and Bad of AI Tools in Novice Programming Education, 3.5)

Novice programmers improved their prompt-engineering skills as they progressed through the semester.

"Since students used the same language model, this result suggests that they improved their prompt-engineering skills over time."

(The Good and Bad of AI Tools in Novice Programming Education, 3.3)

AI tools provided a motivational boost to students when they encountered obstacles in coding.

"receiving feedback from AI tools when they were stuck boosted their self-efficacy and motivation to learn."

(The Good and Bad of AI Tools in Novice Programming Education, 4.2)

RUN LABEL: 13

Prompt: A

Model: ChatGPT 5.2 Auto

Paper: Measuring GitHub Copilot's Impact on Productivity

Output:

None

Here are **five key claims** drawn directly from the paper, each supported by a quote from the text and a clear citation.

Claim

Supporting Quote / Snippet

Citation

Acceptance rate of Copilot suggestions is the strongest predictor of developers' perceived productivity.

"We find acceptance rate (accepted _ per _ shown) most positively predicts users' perception of productivity."

(Measuring GitHub Copilot's Impact on Productivity, *What Drives Perceived Productivity*)

Acceptance rate correlates better with perceived productivity than more detailed contribution or persistence metrics.

"We were surprised to find acceptance rate ... to be better correlated with reported productivity than our measures of persistence."

(Measuring GitHub Copilot's Impact on Productivity, *Conclusions*)

More detailed persistence-based metrics show weaker correlations with perceived productivity than acceptance rate.

"All persistence measures were less well correlated than acceptance rate."

(Measuring GitHub Copilot's Impact on Productivity, *What Drives Perceived Productivity*)

Junior developers report higher productivity gains from Copilot and accept suggestions more frequently than senior developers.

"Junior developers not only report higher productivity gains; they also tend to accept more suggestions."

(Measuring GitHub Copilot's Impact on Productivity, *Experience*)

Experienced developers benefit less in code quality improvements but still gain productivity through flow and reduced repetitive work.

"Experienced developers who are already highly skilled are less likely to write better code with Copilot, but Copilot can assist their productivity in other ways, particularly when engaging with new areas and automating routine work."

(Measuring GitHub Copilot's Impact on Productivity, *Experience*)

RUN LABEL: 14

Prompt: B

Model: ChatGPT 5.2 Auto

Paper: Measuring GitHub Copilot's Impact on Productivity

Output:

None

Claim

Evidence Snippet

Citation (source_id, chunk_id)

Acceptance rate is the strongest usage-based predictor of developers' perceived productivity when using GitHub Copilot.

"We find acceptance rate (accepted _ per _ shown) most positively predicts users' perception of productivity... Of all usage measurements, acceptance rate correlates best with aggregate productivity ($p = 0.24$, $P < 0.0001$)."

(Measuring GitHub Copilot's Impact on Productivity, What Drives Perceived Productivity)

Acceptance rate correlates better with perceived productivity than more detailed contribution or persistence measures.

"Our results support the principle that acceptance rate can be used for coarse-grained monitoring... This ratio of shown suggestions being accepted correlates better than more detailed measures of contribution."

(Measuring GitHub Copilot's Impact on Productivity, Results)

Persistence-based metrics of accepted code are less correlated with perceived productivity than acceptance rate.

"All persistence measures were less well correlated than acceptance rate."

(Measuring GitHub Copilot's Impact on Productivity, What Drives Perceived Productivity)

Junior developers report higher productivity gains and accept more suggestions than more experienced developers.

"Junior developers not only report higher productivity gains; they also tend to accept more suggestions."

(Measuring GitHub Copilot's Impact on Productivity, Experience)

Acceptance rate varies significantly by time of day and day of week, with higher rates outside typical working hours.

"We observe strong regular patterns in overall acceptance rate... weekends... average acceptance rate... 23.5%... typical working hours... much lower at 21.2%."

(Measuring GitHub Copilot's Impact on Productivity, Variation over Time)

RUN LABEL: 15

Prompt: A

Model: Gemini 3 Fast

Paper: Measuring GitHub Copilot's Impact on Productivity

Output:

None

Based on the paper provided, here are five key claims regarding the impact of AI tools (specifically GitHub Copilot) on software engineering:

Key Claims: AI Tools in Software Engineering

Key Claim

Supporting Quote/Snippet

Citation

Acceptance rate is the best predictor of productivity.

"We find that acceptance rate of shown suggestions is a better predictor of perceived productivity than the alternative measures."

"Measuring GitHub Copilot's Impact on Productivity" (Results section)

Junior developers experience higher gains from AI tools.

"Junior developers not only report higher productivity gains; they also tend to accept more suggestions."

"Measuring GitHub Copilot's Impact on Productivity" (Experience section)

Experienced developers use AI for routine work rather than code quality.

"...experienced developers who are already highly skilled are less likely to write better code with Copilot, but Copilot can assist their productivity in other ways, particularly when engaging with new areas and automating routine work."

"Measuring GitHub Copilot's Impact on Productivity" (Experience section)

Acceptance rate varies by the time of day and week.

"Typical working hours during the week... the average acceptance rate is much lower at 21.2% [compared to the weekend and non-working hours]."

"Measuring GitHub Copilot's Impact on Productivity" (Variation over Time section)

The value of AI lies in goal progress rather than perfect accuracy.

"GitHub Copilot's central value lies not in being the way users enter most of their code. Instead, it lies in helping users to make the best progress toward their goals."

"Measuring GitHub Copilot's Impact on Productivity" (Conclusions section)

RUN LABEL: 16

Prompt: B

Model: Gemini 3 Fast

Paper: Measuring GitHub Copilot's Impact on Productivity

Output:

None

To understand the impact of AI-assisted software engineering, researchers analyzed how developer interactions with GitHub Copilot relate to their own feelings of productivity.

Extracted Claims and Evidence

Claim

Evidence Snippet

Citation (source_id, chunk_id)

Acceptance rate is the strongest predictor of a developer's perceived productivity among usage metrics.

"We find that acceptance rate of shown suggestions is a better predictor of perceived productivity than the alternative measures."

(Paper, 1)

Offline evaluation of code completion systems often fails to reflect real-world performance accurately.

"This metric fell from 90% in offline evaluation to 70% when used online."

(Paper, 1)

Highly experienced developers may derive less improvement in code quality from Copilot than their junior peers.

"Both language proficiency and years of experience negatively predict developers agreeing that Copilot helps them write better code."

(Paper, 2)

Usage patterns for Copilot, specifically acceptance rates, fluctuate based on the time of day and week.

"Acceptance rate typically increases over the board... we distinguish three different time regimes... The weekend... typical non-working hours... typical working hours."

(Paper, 2)

Perceived productivity is best measured as a multidimensional concept rather than a single metric.

"The SPACE framework defines five dimensions of productivity: Satisfaction and well-being, Performance, Activity, Communication and collaboration, and Efficiency and flow."

(Paper, 2)

Paper 1 Prompts

Quick Copy and Paste for:

- [PT Prompt A](#)
- [PT Prompt B](#)
- [CEE Prompt A](#)
- [CEE Prompt B](#)

The Impact of AI on Developer Productivity: Evidence from GitHub Copilot

Method: Paper Triage

Prompt A:

You are given an academic paper about AI tools in software engineering.

Summarize the paper using the following fields:

Contribution

Method

Data

Findings

Limitations

Paper: The Impact of AI on Developer Productivity: Evidence from GitHub Copilot

--- BEGIN PAPER TEXT ---

Abstract Generative AI tools hold promise to increase human productivity. This paper presents results from a controlled experiment with GitHub Copilot, an AI pair programmer. Recruited software developers were asked to implement an HTTP server in JavaScript as quickly as possible. The treatment group, with access to the AI pair programmer, completed the task 55.8% faster than the control group. Observed heterogenous effects show promise for AI pair programmers to help people transition into software development careers. Study Design We conducted a controlled experiment to measure the productivity impact of using GitHub Copilot in programming tasks. The experiment began on May 15, 2022 and ended on June 20, 2022, right before GitHub Copilot became generally available. We recruited 95 professional programmers through Upwork, a freelancing platform. Participation in the experiment was advertised on Upwork as a job posting, looking to recruit freelancer developers. Figures 1 and 2 show (respectively) the job posting and the contract that was sent to participants to sign, in accordance with Upwork's policies. Once participants signed the contract, they were randomly split into control and treatment groups. Figure 3 shows the instructions sent to each group through email. The treated group was instructed to watch a 1-minute video introducing them to GitHub Copilot. In addition to the instructions, they also received an automated email with

installation instructions for GitHub Copilot once granted access to the tool. We verify from telemetry after the experiment that all participants from the treated group have configured GitHub Copilot and accepted recommendations other than five who did not finish the sign up and thus started the experiment without the GitHub Copilot. Both treated and control groups were instructed to complete an entry survey to provide demographic information such as age, gender, location, and educational background. Before we began recruitment, we received approval for the study from the Microsoft Research Ethics Review Board. Participants were instructed to write an HTTP server in JavaScript—the treatment group would use GitHub Copilot to complete the task, while the control group would not. Besides the use of GitHub Copilot in the treated group, participants were unconstrained in their software development—they could use any sources of information as they normally do, such as internet search and Stack Overflow. 3 We calculated two metrics as a measure of performance for each group: task success and task completion time. Task success was measured as the percentage of participants in a group that adequately completed the task. Task completion time was measured as the time from start to end of the task. Using a standardized task provides us with precise measures of performance as it is difficult to measure productivity of software developers. To administer the task, we used GitHub Classroom, a platform for teachers to issue and grade coding assignments. In this way, we accurately measured the timing and completion for each participant. The instructions gave participants a link to a particular GitHub Classroom instance with a single assignment referencing a template repository. When joining the assignment, participants received a personal copy of the template repository, with the task description (shown in Figure 4) and a skeleton codebase for participants to build upon. The creation date and time of that personal copy created a timestamp. Each participant's repository was private to them and visible to the researchers conducting the experiment—but not to other participants. We included a test suite in the repository, comprising twelve checks for submission correctness. If a submission passes, all twelve tests we counted are successfully completed. Participants could see the tests but were unable to alter them. When participants committed and pushed their changes to GitHub, GitHub Classroom ran the test suite on their submission and reported the number of passing tests. Participants could push as often as they pleased, automatically logging a timestamp each time. The time elapsed between the timestamp of repository creation and the timestamp of the first commit to successfully pass all 12 tests was counted as the participant's task completion time. The full history of test suite runs is visible on each repository, enabling researchers to observe partial results for participants that did not fully complete the task. The participants' final compensation is calculated based on their time to completion and the scale we had previously shared with them (shown in Figure 1). 4 After participants had completed the task, we sent them the link to an exit survey. We asked the treatment group how helpful they found GitHub Copilot as they worked on the task, as well as asked them to estimate how much faster they completed the task compared to how long this task would have taken them without using GitHub Copilot. We also asked the control group to estimate the size of the speed gain they would have experienced if they used GitHub Copilot, after showing them a 1-minute demo video. Results A total of 166 offers were sent during the experiment, and 95 were accepted. The 95 developers were randomly assigned into control and treated groups, with 45 in the treated group and 50 in control. Thirty-five developers from both the treated and control groups completed the task and survey. Figure 5 presents the summary statistics of these participants. Most of the participants

are in the age group of 25-34 and come from India and Pakistan. This group of participants is also characterized by relatively lower income (median yearly income between \$10,000-\$19,000) compared to US standards but high education level (the majority have a 4-year degree and above). The group has an average coding experience of 6 years and, on average, reported spending 9 hours on coding in a working day. Figure 6 plots the distribution between time to completion between treated and control groups. Conditioning on completing the task, the average completion time from the treated group is 71.17 minutes and 160.89 minutes for the control group. This represents a 55.8% reduction in completion time. The p-value for the t-test is 0.0017, and a 95% confidence interval for the improvement is between [21%, 89%]. There are four outliers with time to completion above 300 min. All outliers are in the control group, however our results remain robust if these outliers are dropped. This result suggests that Copilot increases average productivity significantly in our experiment population. We also find that the treated group's success rate is 7 percentage points higher than the control group, but the estimate is not statistically significant, with a 95% confidence interval of [-0.11, 0.25]. We then investigate whether this effect is heterogeneous across different dimensions including experience, employment status, income, education and software language preference. We assume the treatment effect is a linear function of the covariates of interest. We apply HorvitzThomson transformation in [Athey and Imbens, 2015] (see also [Banerjee and Duflo, 2003] and [Carneiro et al., 2011])) and then regress the transformed outcome of interest on observables. The estimates in Table 1 report coefficients from this regression. The results show that less experienced developers (years of professional coding), developers with heavy coding load (hours of coding per day), and older developers (developers aged between 25 and 44) benefit more from Copilot. We conducted an exit survey with two questions to learn about the experience of subjects. First, we asked them to estimate how much productivity gain or loss (in percentage term) Copilot provided to them for completing the task. While the control group was not exposed to Copilot during the task, they were given the tutorial video before answering this question so that they are aware of the features of Copilot. Figure 7 presents the distribution of the self-reported productivity gain estimates from the control and treated groups. On average, participants in both treated and control groups estimated a 35% increase in productivity, which is an underestimation compared with the 55.8% increase in their revealed productivity. In the second question, participants were asked the highest monthly price at which they would be interested in getting notified about the release of GitHub Copilot. The intention is to learn about developers' willingness to pay for Copilot as the answer to this question provides an upper bound for the developers' willingness to pay. Figure 8 presents the distribution of the irrelevant price separated for the control and treated groups. The average irrelevant price for the treated group is \$27.25, and the average irrelevant price for the control group is \$16.91, both per month. The difference is statistically significant at the 95% level. This result provides indirect evidence that treated group benefited from Copilot during their task as their willingness to pay is significantly higher than the control group. Discussion This paper presents evidence on the productivity effects of generative AI tools in software development. To the best of our knowledge, it is the first controlled experiment to measure the productivity of AI tools in professional software development. Our results suggest that Copilot has statistically and practically significant impact on productivity: the treated group that has access to GitHub Copilot was able to complete the task 55.8% faster than the control group. Further investigations into

the productivity impacts of AI-powered tools in software development are warranted. This study examines a standardized programming task in an experiment to obtain a precise measure of productivity, instead of a task where developers collaborate on large projects in professional proprietary and/or open-source settings. Productivity benefits may vary across specific tasks and programming languages, so more research is needed to understand how our results generalizes to other tasks. Finally, this study does not examine the effects of AI on code quality. AI assistance can increase code quality if it suggests code better than the programmer writes, or it can reduce quality if the programmer pays less attention to code. The code quality can have performance and security considerations that can change the real-world impact of AI. The heterogeneous effects identified in this study warrant close attention. Our results suggest that less experienced programmers benefit more from Copilot. If this result persists in further studies, the productivity benefits for novice programmers and programmers of older age point to important possibilities for skill initiatives that support job transitions into software development. The economic impacts of these models also warrant further research [Manning et al., 2022], with particular attention on their implications for labor market. In 2021, over 4.6 million people in the United States worked in computer and mathematical occupations,¹ a Bureau of Labor Statistics category that includes computer programmers, data scientists, and statisticians. These workers earned \$464.8 billion or roughly 2% of US GDP. If the results of this study were to be extrapolated to the population level, a 55.8% increase in productivity would imply a significant amount of cost savings in the economy and have a notable impact on GDP growth. It is, as of yet, unclear how such gains would be distributed and how job tasks would change to incorporate AI-powered developer tools. It is important to consider such impacts and to begin research on these implications at the outset [Klinova and Korinek, 2021].

--- END PAPER TEXT ---

The Impact of AI on Developer Productivity: Evidence from GitHub Copilot

Method: Paper Triage

Prompt B:

You are analyzing an academic paper about AI-assisted tools in software engineering.

Produce a structured summary using exactly the following five fields:

Contribution
Method
Data
Findings
Limitations

Constraints / Guardrails

Base all statements strictly on the content of the provided paper.

Do not infer or speculate beyond what is explicitly supported by the text.

If the paper does not clearly specify information for a field, write: "Not specified in the paper."

Be concise, factual, and neutral in tone.

Do not introduce external knowledge or assumptions.

Output only the five fields listed above.

Paper: The Impact of AI on Developer Productivity: Evidence from GitHub Copilot

--- BEGIN PAPER TEXT ---

Abstract Generative AI tools hold promise to increase human productivity. This paper presents results from a controlled experiment with GitHub Copilot, an AI pair programmer. Recruited software developers were asked to implement an HTTP server in JavaScript as quickly as possible. The treatment group, with access to the AI pair programmer, completed the task 55.8% faster than the control group. Observed heterogenous effects show promise for AI pair programmers to help people transition into software development careers. Study Design We conducted a controlled experiment to measure the productivity impact of using GitHub Copilot in programming tasks. The experiment began on May 15, 2022 and ended on June 20, 2022, right before GitHub Copilot became generally available. We recruited 95 professional programmers through Upwork, a freelancing platform. Participation in the experiment was advertised on Upwork as a job posting, looking to recruit freelancer developers. Figures 1 and 2 show (respectively) the job posting and the contract that was sent to participants to sign, in accordance with Upwork's policies. Once participants signed the contract, they were randomly split into control and treatment groups. Figure 3 shows the instructions sent to each group through email. The treated group was instructed to watch a 1-minute video introducing them to

GitHub Copilot. In addition to the instructions, they also received an automated email with installation instructions for GitHub Copilot once granted access to the tool. We verify from telemetry after the experiment that all participants from the treated group have configured GitHub Copilot and accepted recommendations other than five who did not finish the sign up and thus started the experiment without the GitHub Copilot. Both treated and control groups were instructed to complete an entry survey to provide demographic information such as age, gender, location, and educational background. Before we began recruitment, we received approval for the study from the Microsoft Research Ethics Review Board. Participants were instructed to write an HTTP server in JavaScript—the treatment group would use GitHub Copilot to complete the task, while the control group would not. Besides the use of GitHub Copilot in the treated group, participants were unconstrained in their software development—they could use any sources of information as they normally do, such as internet search and Stack Overflow. 3 We calculated two metrics as a measure of performance for each group: task success and task completion time. Task success was measured as the percentage of participants in a group that adequately completed the task. Task completion time was measured as the time from start to end of the task. Using a standardized task provides us with precise measures of performance as it is difficult to measure productivity of software developers. To administer the task, we used GitHub Classroom, a platform for teachers to issue and grade coding assignments. In this way, we accurately measured the timing and completion for each participant. The instructions gave participants a link to a particular GitHub Classroom instance with a single assignment referencing a template repository. When joining the assignment, participants received a personal copy of the template repository, with the task description (shown in Figure 4) and a skeleton codebase for participants to build upon. The creation date and time of that personal copy created a timestamp. Each participant's repository was private to them and visible to the researchers conducting the experiment—but not to other participants. We included a test suite in the repository, comprising twelve checks for submission correctness. If a submission passes, all twelve tests we counted are successfully completed. Participants could see the tests but were unable to alter them. When participants committed and pushed their changes to GitHub, GitHub Classroom ran the test suite on their submission and reported the number of passing tests. Participants could push as often as they pleased, automatically logging a timestamp each time. The time elapsed between the timestamp of repository creation and the timestamp of the first commit to successfully pass all 12 tests was counted as the participant's task completion time. The full history of test suite runs is visible on each repository, enabling researchers to observe partial results for participants that did not fully complete the task. The participants' final compensation is calculated based on their time to completion and the scale we had previously shared with them (shown in Figure 1). 4 After participants had completed the task, we sent them the link to an exit survey. We asked the treatment group how helpful they found GitHub Copilot as they worked on the task, as well as asked them to estimate how much faster they completed the task compared to how long this task would have taken them without using GitHub Copilot. We also asked the control group to estimate the size of the speed gain they would have experienced if they used GitHub Copilot, after showing them a 1-minute demo video. Results A total of 166 offers were sent during the experiment, and 95 were accepted. The 95 developers were randomly assigned into control and treated groups, with 45 in the treated group and 50 in control. Thirty-five developers from both the treated and control groups completed the task and

survey. Figure 5 presents the summary statistics of these participants. Most of the participants are in the age group of 25-34 and come from India and Pakistan. This group of participants is also characterized by relatively lower income (median yearly income between \$10,000-\$19,000) compared to US standards but high education level (the majority have a 4-year degree and above). The group has an average coding experience of 6 years and, on average, reported spending 9 hours on coding in a working day. Figure 6 plots the distribution between time to completion between treated and control groups. Conditioning on completing the task, the average completion time from the treated group is 71.17 minutes and 160.89 minutes for the control group. This represents a 55.8% reduction in completion time. The p-value for the t-test is 0.0017, and a 95% confidence interval for the improvement is between [21%, 89%]. There are four outliers with time to completion above 300 min. All outliers are in the control group, however our results remain robust if these outliers are dropped. This result suggests that Copilot increases average productivity significantly in our experiment population. We also find that the treated group's success rate is 7 percentage points higher than the control group, but the estimate is not statistically significant, with a 95% confidence interval of [-0.11, 0.25]. We then investigate whether this effect is heterogeneous across different dimensions including experience, employment status, income, education and software language preference. We assume the treatment effect is a linear function of the covariates of interest. We apply HorvitzThomson transformation in [Athey and Imbens, 2015] (see also [Banerjee and Duflo, 2003] and [Carneiro et al., 2011])) and then regress the transformed outcome of interest on observables. The estimates in Table 1 report coefficients from this regression. The results show that less experienced developers (years of professional coding), developers with heavy coding load (hours of coding per day), and older developers (developers aged between 25 and 44) benefit more from Copilot. We conducted an exit survey with two questions to learn about the experience of subjects. First, we asked them to estimate how much productivity gain or loss (in percentage term) Copilot provided to them for completing the task. While the control group was not exposed to Copilot during the task, they were given the tutorial video before answering this question so that they are aware of the features of Copilot. Figure 7 presents the distribution of the self-reported productivity gain estimates from the control and treated groups. On average, participants in both treated and control groups estimated a 35% increase in productivity, which is an underestimation compared with the 55.8% increase in their revealed productivity. In the second question, participants were asked the highest monthly price at which they would be interested in getting notified about the release of GitHub Copilot. The intention is to learn about developers' willingness to pay for Copilot as the answer to this question provides an upper bound for the developers' willingness to pay. Figure 8 presents the distribution of the irrelevant price separated for the control and treated groups. The average irrelevant price for the treated group is \$27.25, and the average irrelevant price for the control group is \$16.91, both per month. The difference is statistically significant at the 95% level. This result provides indirect evidence that treated group benefited from Copilot during their task as their willingness to pay is significantly higher than the control group. Discussion This paper presents evidence on the productivity effects of generative AI tools in software development. To the best of our knowledge, it is the first controlled experiment to measure the productivity of AI tools in professional software development. Our results suggest that Copilot has statistically and practically significant impact on productivity: the treated group that has access to GitHub Copilot

was able to complete the task 55.8% faster than the control group. Further investigations into the productivity impacts of AI-powered tools in software development are warranted. This study examines a standardized programming task in an experiment to obtain a precise measure of productivity, instead of a task where developers collaborate on large projects in professional proprietary and/or open-source settings. Productivity benefits may vary across specific tasks and programming languages, so more research is needed to understand how our results generalizes to other tasks. Finally, this study does not examine the effects of AI on code quality. AI assistance can increase code quality if it suggests code better than the programmer writes, or it can reduce quality if the programmer pays less attention to code. The code quality can have performance and security considerations that can change the real-world impact of AI. The heterogeneous effects identified in this study warrant close attention. Our results suggest that less experienced programmers benefit more from Copilot. If this result persists in further studies, the productivity benefits for novice programmers and programmers of older age point to important possibilities for skill initiatives that support job transitions into software development. The economic impacts of these models also warrant further research [Manning et al., 2022], with particular attention on their implications for labor market. In 2021, over 4.6 million people in the United States worked in computer and mathematical occupations,¹ a Bureau of Labor Statistics category that includes computer programmers, data scientists, and statisticians. These workers earned \$464.8 billion or roughly 2% of US GDP. If the results of this study were to be extrapolated to the population level, a 55.8% increase in productivity would imply a significant amount of cost savings in the economy and have a notable impact on GDP growth. It is, as of yet, unclear how such gains would be distributed and how job tasks would change to incorporate AI-powered developer tools. It is important to consider such impacts and to begin research on these implications at the outset [Klinova and Korinek, 2021].

--- END PAPER TEXT ---

The Impact of AI on Developer Productivity: Evidence from GitHub Copilot

Method: Claim Evidence Extraction

Prompt A:

From the provided paper, extract five key claims about the impact of AI tools on software engineering.

For each claim, include:

The claim

A supporting quote or snippet from the text

A citation

Present your answer in a table.

Paper: The Impact of AI on Developer Productivity: Evidence from GitHub Copilot

--- BEGIN PAPER TEXT ---

Abstract Generative AI tools hold promise to increase human productivity. This paper presents results from a controlled experiment with GitHub Copilot, an AI pair programmer. Recruited software developers were asked to implement an HTTP server in JavaScript as quickly as possible. The treatment group, with access to the AI pair programmer, completed the task 55.8% faster than the control group. Observed heterogenous effects show promise for AI pair programmers to help people transition into software development careers. Study Design We conducted a controlled experiment to measure the productivity impact of using GitHub Copilot in programming tasks. The experiment began on May 15, 2022 and ended on June 20, 2022, right before GitHub Copilot became generally available. We recruited 95 professional programmers through Upwork, a freelancing platform. Participation in the experiment was advertised on Upwork as a job posting, looking to recruit freelancer developers. Figures 1 and 2 show (respectively) the job posting and the contract that was sent to participants to sign, in accordance with Upwork's policies. Once participants signed the contract, they were randomly split into control and treatment groups. Figure 3 shows the instructions sent to each group through email. The treated group was instructed to watch a 1-minute video introducing them to GitHub Copilot. In addition to the instructions, they also received an automated email with installation instructions for GitHub Copilot once granted access to the tool. We verify from telemetry after the experiment that all participants from the treated group have configured GitHub Copilot and accepted recommendations other than five who did not finish the sign up and thus started the experiment without the GitHub Copilot. Both treated and control groups were instructed to complete an entry survey to provide demographic information such as age, gender, location, and educational background. Before we began recruitment, we received approval for the study from the Microsoft Research Ethics Review Board. Participants were

instructed to write an HTTP server in JavaScript—the treatment group would use GitHub Copilot to complete the task, while the control group would not. Besides the use of GitHub Copilot in the treated group, participants were unconstrained in their software development—they could use any sources of information as they normally do, such as internet search and Stack Overflow.³ We calculated two metrics as a measure of performance for each group: task success and task completion time. Task success was measured as the percentage of participants in a group that adequately completed the task. Task completion time was measured as the time from start to end of the task. Using a standardized task provides us with precise measures of performance as it is difficult to measure productivity of software developers. To administer the task, we used GitHub Classroom, a platform for teachers to issue and grade coding assignments. In this way, we accurately measured the timing and completion for each participant. The instructions gave participants a link to a particular GitHub Classroom instance with a single assignment referencing a template repository. When joining the assignment, participants received a personal copy of the template repository, with the task description (shown in Figure 4) and a skeleton codebase for participants to build upon. The creation date and time of that personal copy created a timestamp. Each participant's repository was private to them and visible to the researchers conducting the experiment—but not to other participants. We included a test suite in the repository, comprising twelve checks for submission correctness. If a submission passes, all twelve tests we counted are successfully completed. Participants could see the tests but were unable to alter them. When participants committed and pushed their changes to GitHub, GitHub Classroom ran the test suite on their submission and reported the number of passing tests. Participants could push as often as they pleased, automatically logging a timestamp each time. The time elapsed between the timestamp of repository creation and the timestamp of the first commit to successfully pass all 12 tests was counted as the participant's task completion time. The full history of test suite runs is visible on each repository, enabling researchers to observe partial results for participants that did not fully complete the task. The participants' final compensation is calculated based on their time to completion and the scale we had previously shared with them (shown in Figure 1).⁴ After participants had completed the task, we sent them the link to an exit survey. We asked the treatment group how helpful they found GitHub Copilot as they worked on the task, as well as asked them to estimate how much faster they completed the task compared to how long this task would have taken them without using GitHub Copilot. We also asked the control group to estimate the size of the speed gain they would have experienced if they used GitHub Copilot, after showing them a 1-minute demo video. Results A total of 166 offers were sent during the experiment, and 95 were accepted. The 95 developers were randomly assigned into control and treated groups, with 45 in the treated group and 50 in control. Thirty-five developers from both the treated and control groups completed the task and survey. Figure 5 presents the summary statistics of these participants. Most of the participants are in the age group of 25-34 and come from India and Pakistan. This group of participants is also characterized by relatively lower income (median yearly income between \$10,000-\$19,000) compared to US standards but high education level (the majority have a 4-year degree and above). The group has an average coding experience of 6 years and, on average, reported spending 9 hours on coding in a working day. Figure 6 plots the distribution between time to completion between treated and control groups. Conditioning on completing the task, the average completion time from the treated group is 71.17 minutes and 160.89 minutes for the

control group. This represents a 55.8% reduction in completion time. The p-value for the t-test is 0.0017, and a 95% confidence interval for the improvement is between [21%, 89%]. There are four outliers with time to completion above 300 min. All outliers are in the control group, however our results remain robust if these outliers are dropped. This result suggests that Copilot increases average productivity significantly in our experiment population. We also find that the treated group's success rate is 7 percentage points higher than the control group, but the estimate is not statistically significant, with a 95% confidence interval of [-0.11, 0.25]. We then investigate whether this effect is heterogeneous across different dimensions including experience, employment status, income, education and software language preference. We assume the treatment effect is a linear function of the covariates of interest. We apply HorvitzThomson transformation in [Athey and Imbens, 2015] (see also [Banerjee and Duflo, 2003] and [Carneiro et al., 2011])) and then regress the transformed outcome of interest on observables. The estimates in Table 1 report coefficients from this regression. The results show that less experienced developers (years of professional coding), developers with heavy coding load (hours of coding per day), and older developers (developers aged between 25 and 44) benefit more from Copilot. We conducted an exit survey with two questions to learn about the experience of subjects. First, we asked them to estimate how much productivity gain or loss (in percentage term) Copilot provided to them for completing the task. While the control group was not exposed to Copilot during the task, they were given the tutorial video before answering this question so that they are aware of the features of Copilot. Figure 7 presents the distribution of the self-reported productivity gain estimates from the control and treated groups. On average, participants in both treated and control groups estimated a 35% increase in productivity, which is an underestimation compared with the 55.8% increase in their revealed productivity. In the second question, participants were asked the highest monthly price at which they would be interested in getting notified about the release of GitHub Copilot. The intention is to learn about developers' willingness to pay for Copilot as the answer to this question provides an upper bound for the developers' willingness to pay. Figure 8 presents the distribution of the irrelevant price separated for the control and treated groups. The average irrelevant price for the treated group is \$27.25, and the average irrelevant price for the control group is \$16.91, both per month. The difference is statistically significant at the 95% level. This result provides indirect evidence that treated group benefited from Copilot during their task as their willingness to pay is significantly higher than the control group.

Discussion

This paper presents evidence on the productivity effects of generative AI tools in software development. To the best of our knowledge, it is the first controlled experiment to measure the productivity of AI tools in professional software development. Our results suggest that Copilot has statistically and practically significant impact on productivity: the treated group that has access to GitHub Copilot was able to complete the task 55.8% faster than the control group. Further investigations into the productivity impacts of AI-powered tools in software development are warranted. This study examines a standardized programming task in an experiment to obtain a precise measure of productivity, instead of a task where developers collaborate on large projects in professional proprietary and/or open-source settings. Productivity benefits may vary across specific tasks and programming languages, so more research is needed to understand how our results generalizes to other tasks. Finally, this study does not examine the effects of AI on code quality. AI assistance can increase code quality if it suggests code better than the programmer writes,

or it can reduce quality if the programmer pays less attention to code. The code quality can have performance and security considerations that can change the real-world impact of AI. The heterogeneous effects identified in this study warrant close attention. Our results suggest that less experienced programmers benefit more from Copilot. If this result persists in further studies, the productivity benefits for novice programmers and programmers of older age point to important possibilities for skill initiatives that support job transitions into software development. The economic impacts of these models also warrant further research [Manning et al., 2022], with particular attention on their implications for labor market. In 2021, over 4.6 million people in the United States worked in computer and mathematical occupations,¹ a Bureau of Labor Statistics category that includes computer programmers, data scientists, and statisticians. These workers earned \$464.8 billion or roughly 2% of US GDP. If the results of this study were to be extrapolated to the population level, a 55.8% increase in productivity would imply a significant amount of cost savings in the economy and have a notable impact on GDP growth. It is, as of yet, unclear how such gains would be distributed and how job tasks would change to incorporate AI-powered developer tools. It is important to consider such impacts and to begin research on these implications at the outset [Klinova and Korinek, 2021].

--- END PAPER TEXT ---

The Impact of AI on Developer Productivity: Evidence from GitHub Copilot

Method: Claim Evidence Extraction

Prompt B:

You are extracting claims and evidence from a research paper about AI-assisted software engineering.

Produce a table with exactly five rows and the following columns:

Claim	Evidence Snippet	Citation (source_id, chunk_id)

Constraints / Guardrails

Each claim must be directly supported by the provided evidence snippet.

Evidence snippets must be copied verbatim or closely paraphrased from the text.

Citations must use the format: (source_id, chunk_id).

Do not invent claims, evidence, or citations.

If the paper does not contain enough information to extract five well-supported claims, include fewer claims and explicitly state “Insufficient evidence for additional claims.”

Do not rely on external knowledge.

Paper: The Impact of AI on Developer Productivity: Evidence from GitHub Copilot

--- BEGIN PAPER TEXT ---

Abstract Generative AI tools hold promise to increase human productivity. This paper presents results from a controlled experiment with GitHub Copilot, an AI pair programmer. Recruited software developers were asked to implement an HTTP server in JavaScript as quickly as possible. The treatment group, with access to the AI pair programmer, completed the task 55.8% faster than the control group. Observed heterogenous effects show promise for AI pair programmers to help people transition into software development careers. Study Design We conducted a controlled experiment to measure the productivity impact of using GitHub Copilot in programming tasks. The experiment began on May 15, 2022 and ended on June 20, 2022, right before GitHub Copilot became generally available. We recruited 95 professional programmers through Upwork, a freelancing platform. Participation in the experiment was advertised on Upwork as a job posting, looking to recruit freelancer developers. Figures 1 and 2 show (respectively) the job posting and the contract that was sent to participants to sign, in accordance with Upwork’s policies. Once participants signed the contract, they were randomly split into control and treatment groups. Figure 3 shows the instructions sent to each group through email. The treated group was instructed to watch a 1-minute video introducing them to GitHub Copilot. In addition to the instructions, they also received an automated email with installation instructions for GitHub Copilot once granted access to the tool. We verify from

telemetry after the experiment that all participants from the treated group have configured GitHub Copilot and accepted recommendations other than five who did not finish the sign up and thus started the experiment without the GitHub Copilot. Both treated and control groups were instructed to complete an entry survey to provide demographic information such as age, gender, location, and educational background. Before we began recruitment, we received approval for the study from the Microsoft Research Ethics Review Board. Participants were instructed to write an HTTP server in JavaScript—the treatment group would use GitHub Copilot to complete the task, while the control group would not. Besides the use of GitHub Copilot in the treated group, participants were unconstrained in their software development—they could use any sources of information as they normally do, such as internet search and Stack Overflow.³ We calculated two metrics as a measure of performance for each group: task success and task completion time. Task success was measured as the percentage of participants in a group that adequately completed the task. Task completion time was measured as the time from start to end of the task. Using a standardized task provides us with precise measures of performance as it is difficult to measure productivity of software developers. To administer the task, we used GitHub Classroom, a platform for teachers to issue and grade coding assignments. In this way, we accurately measured the timing and completion for each participant. The instructions gave participants a link to a particular GitHub Classroom instance with a single assignment referencing a template repository. When joining the assignment, participants received a personal copy of the template repository, with the task description (shown in Figure 4) and a skeleton codebase for participants to build upon. The creation date and time of that personal copy created a timestamp. Each participant's repository was private to them and visible to the researchers conducting the experiment—but not to other participants. We included a test suite in the repository, comprising twelve checks for submission correctness. If a submission passes, all twelve tests we counted are successfully completed. Participants could see the tests but were unable to alter them. When participants committed and pushed their changes to GitHub, GitHub Classroom ran the test suite on their submission and reported the number of passing tests. Participants could push as often as they pleased, automatically logging a timestamp each time. The time elapsed between the timestamp of repository creation and the timestamp of the first commit to successfully pass all 12 tests was counted as the participant's task completion time. The full history of test suite runs is visible on each repository, enabling researchers to observe partial results for participants that did not fully complete the task. The participants' final compensation is calculated based on their time to completion and the scale we had previously shared with them (shown in Figure 1).⁴ After participants had completed the task, we sent them the link to an exit survey. We asked the treatment group how helpful they found GitHub Copilot as they worked on the task, as well as asked them to estimate how much faster they completed the task compared to how long this task would have taken them without using GitHub Copilot. We also asked the control group to estimate the size of the speed gain they would have experienced if they used GitHub Copilot, after showing them a 1-minute demo video. Results A total of 166 offers were sent during the experiment, and 95 were accepted. The 95 developers were randomly assigned into control and treated groups, with 45 in the treated group and 50 in control. Thirty-five developers from both the treated and control groups completed the task and survey. Figure 5 presents the summary statistics of these participants. Most of the participants are in the age group of 25-34 and come from India and Pakistan. This group of participants is

also characterized by relatively lower income (median yearly income between \$10,000-\$19,000) compared to US standards but high education level (the majority have a 4-year degree and above). The group has an average coding experience of 6 years and, on average, reported spending 9 hours on coding in a working day. Figure 6 plots the distribution between time to completion between treated and control groups. Conditioning on completing the task, the average completion time from the treated group is 71.17 minutes and 160.89 minutes for the control group. This represents a 55.8% reduction in completion time. The p-value for the t-test is 0.0017, and a 95% confidence interval for the improvement is between [21%, 89%]. There are four outliers with time to completion above 300 min. All outliers are in the control group, however our results remain robust if these outliers are dropped. This result suggests that Copilot increases average productivity significantly in our experiment population. We also find that the treated group's success rate is 7 percentage points higher than the control group, but the estimate is not statistically significant, with a 95% confidence interval of [-0.11, 0.25]. We then investigate whether this effect is heterogeneous across different dimensions including experience, employment status, income, education and software language preference. We assume the treatment effect is a linear function of the covariates of interest. We apply Horvitz-Thomson transformation in [Athey and Imbens, 2015] (see also [Banerjee and Duflo, 2003] and [Carneiro et al., 2011])) and then regress the transformed outcome of interest on observables. The estimates in Table 1 report coefficients from this regression. The results show that less experienced developers (years of professional coding), developers with heavy coding load (hours of coding per day), and older developers (developers aged between 25 and 44) benefit more from Copilot. We conducted an exit survey with two questions to learn about the experience of subjects. First, we asked them to estimate how much productivity gain or loss (in percentage term) Copilot provided to them for completing the task. While the control group was not exposed to Copilot during the task, they were given the tutorial video before answering this question so that they are aware of the features of Copilot. Figure 7 presents the distribution of the self-reported productivity gain estimates from the control and treated groups. On average, participants in both treated and control groups estimated a 35% increase in productivity, which is an underestimation compared with the 55.8% increase in their revealed productivity. In the second question, participants were asked the highest monthly price at which they would be interested in getting notified about the release of GitHub Copilot. The intention is to learn about developers' willingness to pay for Copilot as the answer to this question provides an upper bound for the developers' willingness to pay. Figure 8 presents the distribution of the irrelevant price separated for the control and treated groups. The average irrelevant price for the treated group is \$27.25, and the average irrelevant price for the control group is \$16.91, both per month. The difference is statistically significant at the 95% level. This result provides indirect evidence that treated group benefited from Copilot during their task as their willingness to pay is significantly higher than the control group. Discussion This paper presents evidence on the productivity effects of generative AI tools in software development. To the best of our knowledge, it is the first controlled experiment to measure the productivity of AI tools in professional software development. Our results suggest that Copilot has statistically and practically significant impact on productivity: the treated group that has access to GitHub Copilot was able to complete the task 55.8% faster than the control group. Further investigations into the productivity impacts of AI-powered tools in software development are warranted. This study

examines a standardized programming task in an experiment to obtain a precise measure of productivity, instead of a task where developers collaborate on large projects in professional proprietary and/or open-source settings. Productivity benefits may vary across specific tasks and programming languages, so more research is needed to understand how our results generalizes to other tasks. Finally, this study does not examine the effects of AI on code quality. AI assistance can increase code quality if it suggests code better than the programmer writes, or it can reduce quality if the programmer pays less attention to code. The code quality can have performance and security considerations that can change the real-world impact of AI. The heterogeneous effects identified in this study warrant close attention. Our results suggest that less experienced programmers benefit more from Copilot. If this result persists in further studies, the productivity benefits for novice programmers and programmers of older age point to important possibilities for skill initiatives that support job transitions into software development. The economic impacts of these models also warrant further research [Manning et al., 2022], with particular attention on their implications for labor market. In 2021, over 4.6 million people in the United States worked in computer and mathematical occupations,¹ a Bureau of Labor Statistics category that includes computer programmers, data scientists, and statisticians. These workers earned \$464.8 billion or roughly 2% of US GDP. If the results of this study were to be extrapolated to the population level, a 55.8% increase in productivity would imply a significant amount of cost savings in the economy and have a notable impact on GDP growth. It is, as of yet, unclear how such gains would be distributed and how job tasks would change to incorporate AI-powered developer tools. It is important to consider such impacts and to begin research on these implications at the outset [Klinova and Korinek, 2021].

--- END PAPER TEXT ---

Paper 2 Prompts

Quick Copy and Paste for:

- [PT Prompt A](#)
- [PT Prompt B](#)
- [CEE Prompt A](#)
- [CEE Prompt B](#)

How AI Impacts Skill Formation

Method: Paper Triage

Prompt A:

You are given an academic paper about AI tools in software engineering.

Summarize the paper using the following fields:

Contribution

Method

Data

Findings

Limitations

Paper: How AI Impacts Skill Formation

--- BEGIN PAPER TEXT ---

Abstract AI assistance produces significant productivity gains across professional domains, particularly for novice workers. Yet how this assistance affects the development of skills required to effectively supervise AI remains unclear. Novice workers who rely heavily on AI to complete unfamiliar tasks may compromise their own skill acquisition in the process. We conduct randomized experiments to study how developers gained mastery of a new asynchronous programming library with and without the assistance of AI. We find that AI use impairs conceptual understanding, code reading, and debugging abilities, without delivering significant efficiency gains on average. Participants who fully delegated coding tasks showed some productivity improvements, but at the cost of learning the library. We identify six distinct AI interaction patterns, three of which involve cognitive engagement and preserve learning outcomes even when participants receive AI assistance. Our findings suggest that AI-enhanced productivity is not a shortcut to competence and AI assistance should be carefully adopted into workflows to preserve skill formation – particularly in safety-critical domains. Introduction As AI promises to be a catalyst for automation and productivity in a wide range of applications, from software engineering to entrepreneurship [Dell'Acqua et al., 2023, Peng et al., 2023, Cui et al., 2024, Otis et al., 2024, Brynjolfsson et al., 2025], the impacts of AI on the labor force are not yet

fully understood. Although more workers rely on AI to improve their productivity, it is unclear whether the use of AI assistance in the workplace might hinder core understanding of concepts or prevent the development of skills necessary to supervise automated tasks. Although most studies have focused on the end product of AI assistance (e.g., lines of code written, quality of ideas proposed), an equally important, if not more crucial question is how process of receiving AI assistance impacts workers. As humans rely on AI for skills such as brainstorming, writing, and general critical thinking, the development of these skills may be significantly altered depending on how AI assistance is used. Software engineering, in particular, has been identified as a profession in which AI tools can be readily applied and AI assistance significantly improves productivity in daily tasks [Peng et al., 2023, Cui et al., 2024]. Junior or novice workers, in particular, benefit most from AI assistance when writing code. In high-stakes applications, AI written code may be debugged and tested by humans before a piece of software is ready for deployment. This additional verification that enhances safety is only possible when human engineers themselves have the skills to understand code and identify errors. As AI development progresses, the problem of supervising more and more capable AI systems becomes more difficult if humans have weaker abilities to understand code [Bowman et al., 2022]. When complex software tasks require human-AI collaboration, humans still need to understand the basic concepts of code development even if their software skills are complementary to the strengths of AI [Wang et al., 2020]. The combination of persistent competency requirements in high-stakes settings and demonstrated productivity gains from AI assistance makes software engineering an ideal testbed for studying how AI affects skill formation. We investigate whether using and relying on AI affects the development of software engineering skills [Handa et al., 2025]. Based on the rapid adoption of AI for software engineering, we are motivated by the scenario of engineers acquiring new skills on the job. Although the use of AI tools may improve productivity for these engineers, would they also inhibit skill formation? More specifically, does an AI-assisted task completion workflow prevent engineers from gaining in-depth knowledge about the tools used to complete these tasks? We run randomized experiments that measure skill formation by asking participants to complete coding tasks with a new library that they have not used before. This represents one way in which engineers acquire and learn new skills, since new libraries are frequently introduced in languages such as Python. We then evaluate their competency with the new library. Our main research questions are (1) whether AI improves productivity for a coding task requiring new concepts and skills, and (2) whether this use of AI reduces the level of understanding of these new concepts and skills.

Methods 4.1 Task Selection: Learning Asynchronous Programming with Trio

We prototyped tasks for several different skills that junior software engineers may encounter on the job: from data analysis to plotting. We designed an experiment around the Python Trio library,¹ which is designed for asynchronous concurrency and input-output processing (I/O). This library is less well known than asyncio (according to the number of StackOverflow questions) and involves new concepts (e.g., structured concurrency) beyond just Python fluency. It is also explicitly designed to be easy to use – making it particularly suitable for a learning experiment. Learned in a professional setting when working with large-scale data or software systems. The tasks we created include problem descriptions, starter code, and brief descriptions of the Trio concepts required to complete the task. These tasks are designed to parallel the process of learning to use a new library or new software tool through a brief self-guided tutorial. For example, in software

engineers' on-boarding materials, there is often a description of how to use an internal library and small tasks to build skills with the new library. After several pilot studies, we used the first two tasks in our main study; each task took 10 - 20 minutes during initial testing. The first task is to write a timer that prints every passing second while other functions run. This task introduces the core concepts of nurseries, starting tasks, and running functions concurrently in Trio. The second task involves implementing a record retrieval function that can handle missing record errors in the Trio library. This task introduces concepts such as error handling and memory channels to store results. These two tasks are standalone; we provide sufficient instructions and usage examples so that participants can complete one task without the other. We used an online interview platform with an AI assistant chat interface (Figure 3) for our experiments.

Participants in the AI condition are prompted to use the AI assistant to help them complete the task. The base model used for this assistant is GPT-4o, and the model is prompted to be an intelligent coding assistant. The AI assistant has access to participants' current version of the code and can produce the full, correct code for both tasks directly when prompted. 4.2

Evaluation Design Based on a previous meta-analysis of evaluations in computer science education [Cheng et al., 2022], we identify four types of questions used to assess the mastery of coding skills. Returning to our initial motivation of developing and retaining the skills required for supervising automation, proficiency in some of these areas may be more important than others for the oversight of AI-generated code. The four types of questions we consider are the following.

- Debugging The ability to identify and diagnose errors in code. This skill is crucial for detecting when AI-generated code is incorrect and understanding why it fails.

- Code Reading The ability to read and comprehend what code does. This skill enables humans to understand and verify AI-written code before deployment.

- Code Writing The ability to write or pick the right way to write code. Low-level code writing, like remembering the syntax of functions, will be less important with further integration of AI coding tools than high-level system design.

- Conceptual The ability to understand the core principles behind tools and libraries. Conceptual understanding is critical to assess whether AI-generated code uses appropriate design patterns that adheres to how the library should be used.

The two tasks in our study cover 7 core concepts from the Trio library. We designed a quiz with debugging, code reading, and conceptual questions that cover these 7 concepts. We exclude code writing questions to reduce the impact of syntax errors in our evaluation; these errors can be easily corrected with an AI query or web search. We tested 5 versions (Table 2) of the quiz in user testing and preliminary studies based on item response theory. For example, we ensure that all questions are sufficiently correlated with the overall quiz score, that each question has an appropriate average score, and that the questions are split up such that there is no local item dependence between questions (i.e., participants could not infer the answers to a question by looking at other questions).

The final evaluation we used contained 14 questions for a total of 27 points. We submitted the grading rubric for the quiz in our study pre-registration before running the experiment. 4.3 Study Design We use a between-subjects randomized experiment to test for the effects of using AI in the coding skill formation process. Each participant first completed a warm-up coding task on a coding platform, where they needed to add a border around a list of strings. This Python coding question takes an average of 4 minutes to complete among users of this coding platform. There are no asynchronous concepts in this coding question. No participants have access to AI while completing the warm-up stage. We use this stage to

calibrate the Python familiarity of the participants and to help participants familiarize themselves with the interface. The next stage is the Trio task stage, where participants have a maximum of 35 minutes to complete two coding tasks using Trio in the same coding platform. During this stage, participants in the AI assistance condition (treatment group) had access to coding help through a chat-based AI assistant (Figure 3). All participants are instructed to complete the task as fast as they could. After completing the Trio task, participants completed the evaluation stage where they take the quiz we described in the previous section and complete a survey that consists of demographic and experiential questions after the quiz. In our main study, 52 participants completed the task, 26 for each of the control and treatment groups. For all our pilot studies and the main study, we only recruited participants who self-reported having more than one year of Python experience, code in Python at least once a week, have tried AI coding assistance at least a few times, and have never used the Trio library before (Table 1). We use the coding platform to collect the keystrokes of the users as they code and the transcripts of their interaction with the AI coding assistant in the coding condition. We use Google Forms to collect survey responses from users both before the coding task and after the coding task. Together, these tasks take a maximum time of 1 hour and 15 minutes with an average duration of 58.5 minutes. Participants were recruited through a third party crowd-worker platform and paid a flat rate of 150 USD for the task. Results Figure 6 shows that while using AI to complete our coding task did not significantly improve task completion time, the level of skill formation gained by completing the task, measured by our quiz, is significantly reduced (Cohen $d=0.738$, $p=0.01$). There is a 4.15 point difference between the means of the treatment and control groups. For a 27-point quiz, this translates into a 17% score difference or 2 grade points. Controlling for warm-up task time as a covariate, the treatment effect remains significant (Cohen's $d=0.725$, $p=0.016$). Prior works have presented mixed results on whether AI helps or hinders coding productivity [Peng et al., 2023, Becker et al., 2025]; our study differs from prior results in that it is designed to study how AI affects skill formation while performing a task requiring new knowledge. While we do observe a slightly lower average completion time in the AI group among novice programmers, due to the small group size of the 1-3 year participant group ($n=4$), the difference in task time was not significant. 4 of the 26 participants in the control (No AI) group did not complete the second task within the 35-minute limit, while every participant in the AI condition completed the second task. Our results do not conclusively find a speed up or slow down using AI in this task. Across all levels of prior coding experience, users scored higher on average in the control (no AI) than in the treatment (AI assistance) group (Figure 7). This shows that our choice of tasks and task design did not critically hinge on the participants' experience level of the but presented new skills to be acquired for every experience group. Concept Group Analysis In exploratory data analysis (not pre-registered), the quiz score was decomposed into subareas and question types (Figure 8). Each question in the quiz belonged to exactly one task (e.g., Task 1 or Task 2) and exactly one question type (e.g., Conceptual, Debugging, or Code Reading). For both tasks, there is a gap between the quiz scores between the treatment and control groups. Among the different types of questions, the largest score gap occurs in the debugging questions and the smallest score gap in the code reading questions. This outcome is expected since treatment and control groups may have similar exposure to reading code through the task, but the control group with no access to AI assistance encountered more errors during the task and became more capable at debugging.

Task Experience In further exploratory data analysis, we also find differences in the way participants' experience of completing the study. The control group (No AI) reported higher self-reported learning (on a 7-point scale), while both groups reported high levels of enjoyment in completing the task (Figure 9). In terms of difficulty of the task, Figure 10 shows that although participants in the treatment group (AI Assistance) found the task easier than the control group, both groups found the post-task quiz similarly challenging. **Discussion** Our main finding is that using AI to complete tasks that require a new skill (i.e., knowledge of a new Python library) reduces skill formation. In a randomized controlled trial, participants were assigned to the treatment condition (using an AI assistant, web search, and instructions) or the control condition (completing tasks with web search and instructions alone). The erosion of conceptual understanding, code reading, and debugging skills that we measured among participants using AI assistance suggests that workers acquiring new skills should be mindful of their reliance on AI during the learning process. Among participants who use AI, we find a stark divide in skill formation outcomes between high-scoring interaction patterns (65%-86% quiz score) vs low-scoring interaction patterns (24%-39% quiz score). The high scorers only asked AI conceptual questions instead of code generation or asked for explanations to accompany generated code; these usage patterns demonstrate a high level of cognitive engagement. Contrary to our initial hypothesis, we did not observe a significant performance boost in task completion in our main study. While using AI improved the average completion time of the task, the improvement in efficiency was not significant in our study, despite the AI Assistant being able to generate the complete code solution when prompted. Our qualitative analysis reveals that our finding is largely due to the heterogeneity in how participants decide to use AI during the task. There is a group of participants who relied on AI to generate all the code and never asked conceptual questions or for explanations. This group finished much faster than the control group (19.5 minutes vs 23 minutes), but this group only accounted for around 20% of the participants in the treatment group. Other participants in the AI group who asked a large number of queries (e.g., 15 queries), spent a long time composing queries (e.g., 10 minutes), or asked for follow-up explanations, raised the average task completion time. These contrasting patterns of AI usage suggest that accomplishing a task with new knowledge or skills does not necessarily lead to the same productive gains as tasks that require only existing knowledge. Together, our results suggest that the aggressive incorporation of AI into the workplace can have negative impacts on the professional development workers if they do not remain cognitatively engaged. Given time constraints and organizational pressures, junior developers or other professionals may rely on AI to complete tasks as fast as possible at the cost of real skill development. Furthermore, we found that the biggest difference in test scores is between the debugging questions. This suggests that as companies transition to more AI code writing with human supervision, humans may not possess the necessary skills to validate and debug AI-written code if their skill formation was inhibited by using AI in the first place.

7.1 Future Work

Our work is a first step to understanding the impact of AI assistance on humans in the human-AI collaboration process. We hope that this work will motivate future work that addresses the following limitations:

- **Task Selection:** This study focuses on a single task using a chat-based interface. This should be a lower bound for cognitive offloading since agentic AI coding tools would require even less human participation. In our work, users who relied on AI without thinking performed the worst on the evaluation; a completely agentic tool would create a similar

effect. Future work should investigate the impacts of agentic coding tools on learning outcomes and skill development.

- Task Length: Ideally, skill formation takes place over months to years. We measured skill formation for a specific Python library over a one-hour period. Future work should study real-world skill development through longitudinal measurement of the impacts of AI adoption.
- Participant Realism: While participants in our study were professional or freelance programmers, there was not the same incentive to learn the library as if it were required for their actual job. Future studies should aim at studying the skill acquisition from novice workers within a real company.
- Prompting Skills: We collect self-reported familiarity with AI coding tools, but we do not actually measure differences in prompting techniques. An extension to our work would also involve testing the level of prompting fluency beyond self-report.
- Evaluation Design: Our study measures skill formation through a comprehensive quiz. Other studies could use the completion of another task or design coding as alternative evaluation strategies.
- Human Assistance: We do not include the counterfactual of how skill formation would be impacted by receiving assistance from humans. Since human assistance and feedback takes place in a diverse settings (e.g., classroom, pair programming, code review), future work can compare the effect of feedback from AI vs humans in all these settings on skill formation. For novice workers in software engineering or any other industry, our study can be viewed as a small piece of evidence toward the value of intentional skill development despite the ubiquity of AI tools. Our study demonstrates the benefits of deploying cognitive effort when encountering a learning opportunity to master a new tool even if barriers (e.g., errors) may be encountered in the process of mastery. Exerting cognitive effort can be assisted by AI; beyond the patterns we describe, major LLM services also provide learning modes (e.g., ChatGPT Study Mode, Claude Code Learning / Explanatory mode). Ultimately, to accommodate skill development in the presence of AI, there needs to be a more expansive view of the impacts of AI on workers. Participants in the new AI economy must care not only about productivity gains from AI but also the long-term sustainability of expertise development amid the proliferation of new AI tools.

--- END PAPER TEXT ---

How AI Impacts Skill Formation

Method: Paper Triage

Prompt B:

You are analyzing an academic paper about AI-assisted tools in software engineering.

Produce a structured summary using exactly the following five fields:

Contribution
Method
Data
Findings
Limitations

Constraints / Guardrails

Base all statements strictly on the content of the provided paper.

Do not infer or speculate beyond what is explicitly supported by the text.

If the paper does not clearly specify information for a field, write: "Not specified in the paper."

Be concise, factual, and neutral in tone.

Do not introduce external knowledge or assumptions.

Output only the five fields listed above.

Paper: How AI Impacts Skill Formation

--- BEGIN PAPER TEXT ---

Abstract AI assistance produces significant productivity gains across professional domains, particularly for novice workers. Yet how this assistance affects the development of skills required to effectively supervise AI remains unclear. Novice workers who rely heavily on AI to complete unfamiliar tasks may compromise their own skill acquisition in the process. We conduct randomized experiments to study how developers gained mastery of a new asynchronous programming library with and without the assistance of AI. We find that AI use impairs conceptual understanding, code reading, and debugging abilities, without delivering significant efficiency gains on average. Participants who fully delegated coding tasks showed some productivity improvements, but at the cost of learning the library. We identify six distinct AI interaction patterns, three of which involve cognitive engagement and preserve learning outcomes even when participants receive AI assistance. Our findings suggest that AI-enhanced productivity is not a shortcut to competence and AI assistance should be carefully adopted into workflows to preserve skill formation – particularly in safety-critical domains. Introduction As AI promises to be a catalyst for automation and productivity in a wide range of applications, from software engineering to entrepreneurship [Dell'Acqua et al., 2023, Peng et al., 2023, Cui et al.,

2024, Otis et al., 2024, Brynjolfsson et al., 2025], the impacts of AI on the labor force are not yet fully understood. Although more workers rely on AI to improve their productivity, it is unclear whether the use of AI assistance in the workplace might hinder core understanding of concepts or prevent the development of skills necessary to supervise automated tasks. Although most studies have focused on the end product of AI assistance (e.g., lines of code written, quality of ideas proposed), an equally important, if not more crucial question is how process of receiving AI assistance impacts workers. As humans rely on AI for skills such as brainstorming, writing, and general critical thinking, the development of these skills may be significantly altered depending on how AI assistance is used. Software engineering, in particular, has been identified as a profession in which AI tools can be readily applied and AI assistance significantly improves productivity in daily tasks [Peng et al., 2023, Cui et al., 2024]. Junior or novice workers, in particular, benefit most from AI assistance when writing code. In high-stakes applications, AI written code may be debugged and tested by humans before a piece of software is ready for deployment. This additional verification that enhances safety is only possible when human engineers themselves have the skills to understand code and identify errors. As AI development progresses, the problem of supervising more and more capable AI systems becomes more difficult if humans have weaker abilities to understand code [Bowman et al., 2022]. When complex software tasks require human-AI collaboration, humans still need to understand the basic concepts of code development even if their software skills are complementary to the strengths of AI [Wang et al., 2020]. The combination of persistent competency requirements in high-stakes settings and demonstrated productivity gains from AI assistance makes software engineering an ideal testbed for studying how AI affects skill formation. We investigate whether using and relying on AI affects the development of software engineering skills [Handa et al., 2025]. Based on the rapid adoption of AI for software engineering, we are motivated by the scenario of engineers acquiring new skills on the job. Although the use of AI tools may improve productivity for these engineers, would they also inhibit skill formation? More specifically, does an AI-assisted task completion workflow prevent engineers from gaining in-depth knowledge about the tools used to complete these tasks? We run randomized experiments that measure skill formation by asking participants to complete coding tasks with a new library that they have not used before. This represents one way in which engineers acquire and learn new skills, since new libraries are frequently introduced in languages such as Python. We then evaluate their competency with the new library. Our main research questions are (1) whether AI improves productivity for a coding task requiring new concepts and skills, and (2) whether this use of AI reduces the level of understanding of these new concepts and skills.

Methods

4.1 Task Selection: Learning Asynchronous Programming with Trio

We prototyped tasks for several different skills that junior software engineers may encounter on the job: from data analysis to plotting. We designed an experiment around the Python Trio library,¹ which is designed for asynchronous concurrency and input-output processing (I/O). This library is less well known than asyncio (according to the number of StackOverflow questions) and involves new concepts (e.g., structured concurrency) beyond just Python fluency. It is also explicitly designed to be easy to use – making it particularly suitable for a learning experiment. Learned in a professional setting when working with large-scale data or software systems. The tasks we created include problem descriptions, starter code, and brief descriptions of the Trio concepts required to complete the task. These tasks are designed to parallel the process of learning to use a new

library or new software tool through a brief self-guided tutorial. For example, in software engineers' on-boarding materials, there is often a description of how to use an internal library and small tasks to build skills with the new library. After several pilot studies, we used the first two tasks in our main study; each task took 10 - 20 minutes during initial testing. The first task is to write a timer that prints every passing second while other functions run. This task introduces the core concepts of nurseries, starting tasks, and running functions concurrently in Trio. The second task involves implementing a record retrieval function that can handle missing record errors in the Trio library. This task introduces concepts such as error handling and memory channels to store results. These two tasks are standalone; we provide sufficient instructions and usage examples so that participants can complete one task without the other. We used an online interview platform with an AI assistant chat interface (Figure 3) for our experiments.

Participants in the AI condition are prompted to use the AI assistant to help them complete the task. The base model used for this assistant is GPT-4o, and the model is prompted to be an intelligent coding assistant. The AI assistant has access to participants' current version of the code and can produce the full, correct code for both tasks directly when prompted.

4.2 Evaluation Design Based on a previous meta-analysis of evaluations in computer science education [Cheng et al., 2022], we identify four types of questions used to assess the mastery of coding skills. Returning to our initial motivation of developing and retaining the skills required for supervising automation, proficiency in some of these areas may be more important than others for the oversight of AI-generated code. The four types of questions we consider are the following.

- Debugging The ability to identify and diagnose errors in code. This skill is crucial for detecting when AI-generated code is incorrect and understanding why it fails.
- Code Reading The ability to read and comprehend what code does. This skill enables humans to understand and verify AI-written code before deployment.
- Code Writing The ability to write or pick the right way to write code. Low-level code writing, like remembering the syntax of functions, will be less important with further integration of AI coding tools than high-level system design.
- Conceptual The ability to understand the core principles behind tools and libraries. Conceptual understanding is critical to assess whether AI-generated code uses appropriate design patterns that adheres to how the library should be used.

The two tasks in our study cover 7 core concepts from the Trio library. We designed a quiz with debugging, code reading, and conceptual questions that cover these 7 concepts. We exclude code writing questions to reduce the impact of syntax errors in our evaluation; these errors can be easily corrected with an AI query or web search. We tested 5 versions (Table 2) of the quiz in user testing and preliminary studies based on item response theory. For example, we ensure that all questions are sufficiently correlated with the overall quiz score, that each question has an appropriate average score, and that the questions are split up such that there is no local item dependence between questions (i.e., participants could not infer the answers to a question by looking at other questions). The final evaluation we used contained 14 questions for a total of 27 points. We submitted the grading rubric for the quiz in our study pre-registration before running the experiment.

4.3 Study Design We use a between-subjects randomized experiment to test for the effects of using AI in the coding skill formation process. Each participant first completed a warm-up coding task on a coding platform, where they needed to add a border around a list of strings. This Python coding question takes an average of 4 minutes to complete among users of this coding platform. There are no asynchronous concepts in this coding question. No

participants have access to AI while completing the warm-up stage. We use this stage to calibrate the Python familiarity of the participants and to help participants familiarize themselves with the interface. The next stage is the Trio task stage, where participants have a maximum of 35 minutes to complete two coding tasks using Trio in the same coding platform. During this stage, participants in the AI assistance condition (treatment group) had access to coding help through a chat-based AI assistant (Figure 3). All participants are instructed to complete the task as fast as they could. After completing the Trio task, participants completed the evaluation stage where they take the quiz we described in the previous section and complete a survey that consists of demographic and experiential questions after the quiz. In our main study, 52 participants completed the task, 26 for each of the control and treatment groups. For all our pilot studies and the main study, we only recruited participants who self-reported having more than one year of Python experience, code in Python at least once a week, have tried AI coding assistance at least a few times, and have never used the Trio library before (Table 1). We use the coding platform to collect the keystrokes of the users as they code and the transcripts of their interaction with the AI coding assistant in the coding condition. We use Google Forms to collect survey responses from users both before the coding task and after the coding task. Together, these tasks take a maximum time of 1 hour and 15 minutes with an average duration of 58.5 minutes. Participants were recruited through a third party crowd-worker platform and paid a flat rate of 150 USD for the task. Results Figure 6 shows that while using AI to complete our coding task did not significantly improve task completion time, the level of skill formation gained by completing the task, measured by our quiz, is significantly reduced (Cohen $d=0.738$, $p=0.01$). There is a 4.15 point difference between the means of the treatment and control groups. For a 27-point quiz, this translates into a 17% score difference or 2 grade points. Controlling for warm-up task time as a covariate, the treatment effect remains significant (Cohen's $d=0.725$, $p=0.016$). Prior works have presented mixed results on whether AI helps or hinders coding productivity [Peng et al., 2023, Becker et al., 2025]; our study differs from prior results in that it is designed to study how AI affects skill formation while performing a task requiring new knowledge. While we do observe a slightly lower average completion time in the AI group among novice programmers, due to the small group size of the 1-3 year participant group ($n=4$), the difference in task time was not significant. 4 of the 26 participants in the control (No AI) group did not complete the second task within the 35-minute limit, while every participant in the AI condition completed the second task. Our results do not conclusively find a speed up or slow down using AI in this task. Across all levels of prior coding experience, users scored higher on average in the control (no AI) than in the treatment (AI assistance) group (Figure 7). This shows that our choice of tasks and task design did not critically hinge on the participants' experience level of the but presented new skills to be acquired for every experience group. Concept Group Analysis In exploratory data analysis (not pre-registered), the quiz score was decomposed into subareas and question types (Figure 8). Each question in the quiz belonged to exactly one task (e.g., Task 1 or Task 2) and exactly one question type (e.g., Conceptual, Debugging, or Code Reading). For both tasks, there is a gap between the quiz scores between the treatment and control groups. Among the different types of questions, the largest score gap occurs in the debugging questions and the smallest score gap in the code reading questions. This outcome is expected since treatment and control groups may have similar exposure to reading code through the task, but the control group with no access to AI

assistance encountered more errors during the task and became more capable at debugging.

Task Experience In further exploratory data analysis, we also find differences in the way participants' experience of completing the study. The control group (No AI) reported higher self-reported learning (on a 7-point scale), while both groups reported high levels of enjoyment in completing the task (Figure 9). In terms of difficulty of the task, Figure 10 shows that although participants in the treatment group (AI Assistance) found the task easier than the control group, both groups found the post-task quiz similarly challenging. **Discussion** Our main finding is that using AI to complete tasks that require a new skill (i.e., knowledge of a new Python library) reduces skill formation. In a randomized controlled trial, participants were assigned to the treatment condition (using an AI assistant, web search, and instructions) or the control condition (completing tasks with web search and instructions alone). The erosion of conceptual understanding, code reading, and debugging skills that we measured among participants using AI assistance suggests that workers acquiring new skills should be mindful of their reliance on AI during the learning process. Among participants who use AI, we find a stark divide in skill formation outcomes between high-scoring interaction patterns (65%-86% quiz score) vs low-scoring interaction patterns (24%-39% quiz score). The high scorers only asked AI conceptual questions instead of code generation or asked for explanations to accompany generated code; these usage patterns demonstrate a high level of cognitive engagement. Contrary to our initial hypothesis, we did not observe a significant performance boost in task completion in our main study. While using AI improved the average completion time of the task, the improvement in efficiency was not significant in our study, despite the AI Assistant being able to generate the complete code solution when prompted. Our qualitative analysis reveals that our finding is largely due to the heterogeneity in how participants decide to use AI during the task. There is a group of participants who relied on AI to generate all the code and never asked conceptual questions or for explanations. This group finished much faster than the control group (19.5 minutes vs 23 minutes), but this group only accounted for around 20% of the participants in the treatment group. Other participants in the AI group who asked a large number of queries (e.g., 15 queries), spent a long time composing queries (e.g., 10 minutes), or asked for follow-up explanations, raised the average task completion time. These contrasting patterns of AI usage suggest that accomplishing a task with new knowledge or skills does not necessarily lead to the same productive gains as tasks that require only existing knowledge. Together, our results suggest that the aggressive incorporation of AI into the workplace can have negative impacts on the professional development workers if they do not remain cognitatively engaged. Given time constraints and organizational pressures, junior developers or other professionals may rely on AI to complete tasks as fast as possible at the cost of real skill development. Furthermore, we found that the biggest difference in test scores is between the debugging questions. This suggests that as companies transition to more AI code writing with human supervision, humans may not possess the necessary skills to validate and debug AI-written code if their skill formation was inhibited by using AI in the first place.

7.1 Future Work

Our work is a first step to understanding the impact of AI assistance on humans in the human-AI collaboration process. We hope that this work will motivate future work that addresses the following limitations:

- **Task Selection:** This study focuses on a single task using a chat-based interface. This should be a lower bound for cognitive offloading since agentic AI coding tools would require even less human participation. In our work, users who relied on AI without

thinking performed the worst on the evaluation; a completely agentic tool would create a similar effect. Future work should investigate the impacts of agentic coding tools on learning outcomes and skill development.

- Task Length: Ideally, skill formation takes place over months to years. We measured skill formation for a specific Python library over a one-hour period. Future work should study real-world skill development through longitudinal measurement of the impacts of AI adoption.
- Participant Realism: While participants in our study were professional or freelance programmers, there was not the same incentive to learn the library as if it were required for their actual job. Future studies should aim at studying the skill acquisition from novice workers within a real company.
- Prompting Skills: We collect self-reported familiarity with AI coding tools, but we do not actually measure differences in prompting techniques. An extension to our work would also involve testing the level of prompting fluency beyond self-report.
- Evaluation Design: Our study measures skill formation through a comprehensive quiz. Other studies could use the completion of another task or design coding as alternative evaluation strategies.
- Human Assistance: We do not include the counterfactual of how skill formation would be impacted by receiving assistance from humans. Since human assistance and feedback takes place in a diverse settings (e.g., classroom, pair programming, code review), future work can compare the effect of feedback from AI vs humans in all these settings on skill formation. For novice workers in software engineering or any other industry, our study can be viewed as a small piece of evidence toward the value of intentional skill development despite the ubiquity of AI tools. Our study demonstrates the benefits of deploying cognitive effort when encountering a learning opportunity to master a new tool even if barriers (e.g., errors) may be encountered in the process of mastery. Exerting cognitive effort can be assisted by AI; beyond the patterns we describe, major LLM services also provide learning modes (e.g., ChatGPT Study Mode, Claude Code Learning / Explanatory mode). Ultimately, to accommodate skill development in the presence of AI, there needs to be a more expansive view of the impacts of AI on workers. Participants in the new AI economy must care not only about productivity gains from AI but also the long-term sustainability of expertise development amid the proliferation of new AI tools.

--- END PAPER TEXT ---

How AI Impacts Skill Formation

Method: Claim Evidence Extraction

Prompt A:

From the provided paper, extract five key claims about the impact of AI tools on software engineering.

For each claim, include:

The claim

A supporting quote or snippet from the text

A citation

Present your answer in a table.

Paper: How AI Impacts Skill Formation

--- BEGIN PAPER TEXT ---

Abstract AI assistance produces significant productivity gains across professional domains, particularly for novice workers. Yet how this assistance affects the development of skills required to effectively supervise AI remains unclear. Novice workers who rely heavily on AI to complete unfamiliar tasks may compromise their own skill acquisition in the process. We conduct randomized experiments to study how developers gained mastery of a new asynchronous programming library with and without the assistance of AI. We find that AI use impairs conceptual understanding, code reading, and debugging abilities, without delivering significant efficiency gains on average. Participants who fully delegated coding tasks showed some productivity improvements, but at the cost of learning the library. We identify six distinct AI interaction patterns, three of which involve cognitive engagement and preserve learning outcomes even when participants receive AI assistance. Our findings suggest that AI-enhanced productivity is not a shortcut to competence and AI assistance should be carefully adopted into workflows to preserve skill formation – particularly in safety-critical domains. Introduction As AI promises to be a catalyst for automation and productivity in a wide range of applications, from software engineering to entrepreneurship [Dell'Acqua et al., 2023, Peng et al., 2023, Cui et al., 2024, Otis et al., 2024, Brynjolfsson et al., 2025], the impacts of AI on the labor force are not yet fully understood. Although more workers rely on AI to improve their productivity, it is unclear whether the use of AI assistance in the workplace might hinder core understanding of concepts or prevent the development of skills necessary to supervise automated tasks. Although most studies have focused on the end product of AI assistance (e.g., lines of code written, quality of ideas proposed), an equally important, if not more crucial question is how process of receiving AI assistance impacts workers. As humans rely on AI for skills such as brainstorming, writing, and general critical thinking, the development of these skills may be significantly altered

depending on how AI assistance is used. Software engineering, in particular, has been identified as a profession in which AI tools can be readily applied and AI assistance significantly improves productivity in daily tasks [Peng et al., 2023, Cui et al., 2024]. Junior or novice workers, in particular, benefit most from AI assistance when writing code. In high-stakes applications, AI written code may be debugged and tested by humans before a piece of software is ready for deployment. This additional verification that enhances safety is only possible when human engineers themselves have the skills to understand code and identify errors. As AI development progresses, the problem of supervising more and more capable AI systems becomes more difficult if humans have weaker abilities to understand code [Bowman et al., 2022]. When complex software tasks require human-AI collaboration, humans still need to understand the basic concepts of code development even if their software skills are complementary to the strengths of AI [Wang et al., 2020]. The combination of persistent competency requirements in high-stakes settings and demonstrated productivity gains from AI assistance makes software engineering an ideal testbed for studying how AI affects skill formation. We investigate whether using and relying on AI affects the development of software engineering skills [Handa et al., 2025]. Based on the rapid adoption of AI for software engineering, we are motivated by the scenario of engineers acquiring new skills on the job. Although the use of AI tools may improve productivity for these engineers, would they also inhibit skill formation? More specifically, does an AI-assisted task completion workflow prevent engineers from gaining in-depth knowledge about the tools used to complete these tasks? We run randomized experiments that measure skill formation by asking participants to complete coding tasks with a new library that they have not used before. This represents one way in which engineers acquire and learn new skills, since new libraries are frequently introduced in languages such as Python. We then evaluate their competency with the new library. Our main research questions are (1) whether AI improves productivity for a coding task requiring new concepts and skills, and (2) whether this use of AI reduces the level of understanding of these new concepts and skills.

Methods 4.1 Task Selection: Learning Asynchronous Programming with Trio

We prototyped tasks for several different skills that junior software engineers may encounter on the job: from data analysis to plotting. We designed an experiment around the Python Trio library,¹ which is designed for asynchronous concurrency and input-output processing (I/O). This library is less well known than asyncio (according to the number of StackOverflow questions) and involves new concepts (e.g., structured concurrency) beyond just Python fluency. It is also explicitly designed to be easy to use – making it particularly suitable for a learning experiment. Learned in a professional setting when working with large-scale data or software systems. The tasks we created include problem descriptions, starter code, and brief descriptions of the Trio concepts required to complete the task. These tasks are designed to parallel the process of learning to use a new library or new software tool through a brief self-guided tutorial. For example, in software engineers' on-boarding materials, there is often a description of how to use an internal library and small tasks to build skills with the new library. After several pilot studies, we used the first two tasks in our main study; each task took 10 - 20 minutes during initial testing. The first task is to write a timer that prints every passing second while other functions run. This task introduces the core concepts of nurseries, starting tasks, and running functions concurrently in Trio. The second task involves implementing a record retrieval function that can handle missing record errors in the Trio library. This task introduces concepts such as error handling and memory

channels to store results. These two tasks are standalone; we provide sufficient instructions and usage examples so that participants can complete one task without the other. We used an online interview platform with an AI assistant chat interface (Figure 3) for our experiments.

Participants in the AI condition are prompted to use the AI assistant to help them complete the task. The base model used for this assistant is GPT-4o, and the model is prompted to be an intelligent coding assistant. The AI assistant has access to participants' current version of the code and can produce the full, correct code for both tasks directly when prompted.

4.2 Evaluation Design Based on a previous meta-analysis of evaluations in computer science education [Cheng et al., 2022], we identify four types of questions used to assess the mastery of coding skills. Returning to our initial motivation of developing and retaining the skills required for supervising automation, proficiency in some of these areas may be more important than others for the oversight of AI-generated code. The four types of questions we consider are the following.

- Debugging The ability to identify and diagnose errors in code. This skill is crucial for detecting when AI-generated code is incorrect and understanding why it fails.
- Code Reading The ability to read and comprehend what code does. This skill enables humans to understand and verify AI-written code before deployment.
- Code Writing The ability to write or pick the right way to write code. Low-level code writing, like remembering the syntax of functions, will be less important with further integration of AI coding tools than high-level system design.
- Conceptual The ability to understand the core principles behind tools and libraries. Conceptual understanding is critical to assess whether AI-generated code uses appropriate design patterns that adheres to how the library should be used.

The two tasks in our study cover 7 core concepts from the Trio library. We designed a quiz with debugging, code reading, and conceptual questions that cover these 7 concepts. We exclude code writing questions to reduce the impact of syntax errors in our evaluation; these errors can be easily corrected with an AI query or web search. We tested 5 versions (Table 2) of the quiz in user testing and preliminary studies based on item response theory. For example, we ensure that all questions are sufficiently correlated with the overall quiz score, that each question has an appropriate average score, and that the questions are split up such that there is no local item dependence between

questions (i.e., participants could not infer the answers to a question by looking at other questions). The final evaluation we used contained 14 questions for a total of 27 points. We submitted the grading rubric for the quiz in our study pre-registration before running the experiment.

4.3 Study Design We use a between-subjects randomized experiment to test for the effects of using AI in the coding skill formation process. Each participant first completed a warm-up coding task on a coding platform, where they needed to add a border around a list of strings. This Python coding question takes an average of 4 minutes to complete among users of this coding platform. There are no asynchronous concepts in this coding question. No participants have access to AI while completing the warm-up stage. We use this stage to calibrate the Python familiarity of the participants and to help participants familiarize themselves with the interface. The next stage is the Trio task stage, where participants have a maximum of 35 minutes to complete two coding tasks using Trio in the same coding platform. During this stage, participants in the AI assistance condition (treatment group) had access to coding help through a chat-based AI assistant (Figure 3). All participants are instructed to complete the task as fast as they could. After completing the Trio task, participants completed the evaluation stage where they take the quiz we described in the previous section and complete a survey that

consists of demographic and experiential questions after the quiz. In our main study, 52 participants completed the task, 26 for each of the control and treatment groups. For all our pilot studies and the main study, we only recruited participants who self-reported having more than one year of Python experience, code in Python at least once a week, have tried AI coding assistance at least a few times, and have never used the Trio library before (Table 1). We use the coding platform to collect the keystrokes of the users as they code and the transcripts of their interaction with the AI coding assistant in the coding condition. We use Google Forms to collect survey responses from users both before the coding task and after the coding task. Together, these tasks take a maximum time of 1 hour and 15 minutes with an average duration of 58.5 minutes. Participants were recruited through a third party crowd-worker platform and paid a flat rate of 150 USD for the task. Results Figure 6 shows that while using AI to complete our coding task did not significantly improve task completion time, the level of skill formation gained by completing the task, measured by our quiz, is significantly reduced (Cohen $d=0.738$, $p=0.01$). There is a 4.15 point difference between the means of the treatment and control groups. For a 27-point quiz, this translates into a 17% score difference or 2 grade points. Controlling for warm-up task time as a covariate, the treatment effect remains significant (Cohen's $d=0.725$, $p=0.016$). Prior works have presented mixed results on whether AI helps or hinders coding productivity [Peng et al., 2023, Becker et al., 2025]; our study differs from prior results in that it is designed to study how AI affects skill formation while performing a task requiring new knowledge. While we do observe a slightly lower average completion time in the AI group among novice programmers, due to the small group size of the 1-3 year participant group ($n=4$), the difference in task time was not significant. 4 of the 26 participants in the control (No AI) group did not complete the second task within the 35-minute limit, while every participant in the AI condition completed the second task. Our results do not conclusively find a speed up or slow down using AI in this task. Across all levels of prior coding experience, users scored higher on average in the control (no AI) than in the treatment (AI assistance) group (Figure 7). This shows that our choice of tasks and task design did not critically hinge on the participants' experience level of the but presented new skills to be acquired for every experience group. Concept Group Analysis In exploratory data analysis (not pre-registered), the quiz score was decomposed into subareas and question types (Figure 8). Each question in the quiz belonged to exactly one task (e.g., Task 1 or Task 2) and exactly one question type (e.g., Conceptual, Debugging, or Code Reading). For both tasks, there is a gap between the quiz scores between the treatment and control groups. Among the different types of questions, the largest score gap occurs in the debugging questions and the smallest score gap in the code reading questions. This outcome is expected since treatment and control groups may have similar exposure to reading code through the task, but the control group with no access to AI assistance encountered more errors during the task and became more capable at debugging. Task Experience In further exploratory data analysis, we also find differences in the way participants' experience of completing the study. The control group (No AI) reported higher self-reported learning (on a 7-point scale), while both groups reported high levels of enjoyment in completing the task (Figure 9). In terms of difficulty of the task, Figure 10 shows that although participants in the treatment group (AI Assistance) found the task easier than the control group, both groups found the post-task quiz similarly challenging. Discussion Our main finding is that using AI to complete tasks that require a new skill (i.e., knowledge of a new Python library)

reduces skill formation. In a randomized controlled trial, participants were assigned to the treatment condition (using an AI assistant, web search, and instructions) or the control condition (completing tasks with web search and instructions alone). The erosion of conceptual understanding, code reading, and debugging skills that we measured among participants using AI assistance suggests that workers acquiring new skills should be mindful of their reliance on AI during the learning process. Among participants who use AI, we find a stark divide in skill formation outcomes between high-scoring interaction patterns (65%-86% quiz score) vs low-scoring interaction patterns (24%-39% quiz score). The high scorers only asked AI conceptual questions instead of code generation or asked for explanations to accompany generated code; these usage patterns demonstrate a high level of cognitive engagement. Contrary to our initial hypothesis, we did not observe a significant performance boost in task completion in our main study. While using AI improved the average completion time of the task, the improvement in efficiency was not significant in our study, despite the AI Assistant being able to generate the complete code solution when prompted. Our qualitative analysis reveals that our finding is largely due to the heterogeneity in how participants decide to use AI during the task. There is a group of participants who relied on AI to generate all the code and never asked conceptual questions or for explanations. This group finished much faster than the control group (19.5 minutes vs 23 minutes), but this group only accounted for around 20% of the participants in the treatment group. Other participants in the AI group who asked a large number of queries (e.g., 15 queries), spent a long time composing queries (e.g., 10 minutes), or asked for follow-up explanations, raised the average task completion time. These contrasting patterns of AI usage suggest that accomplishing a task with new knowledge or skills does not necessarily lead to the same productive gains as tasks that require only existing knowledge. Together, our results suggest that the aggressive incorporation of AI into the workplace can have negative impacts on the professional development workers if they do not remain cognitatively engaged. Given time constraints and organizational pressures, junior developers or other professionals may rely on AI to complete tasks as fast as possible at the cost of real skill development. Furthermore, we found that the biggest difference in test scores is between the debugging questions. This suggests that as companies transition to more AI code writing with human supervision, humans may not possess the necessary skills to validate and debug AI-written code if their skill formation was inhibited by using AI in the first place.

7.1 Future Work

Our work is a first step to understanding the impact of AI assistance on humans in the human-AI collaboration process. We hope that this work will motivate future work that addresses the following limitations:

- **Task Selection:** This study focuses on a single task using a chat-based interface. This should be a lower bound for cognitive offloading since agentic AI coding tools would require even less human participation. In our work, users who relied on AI without thinking performed the worst on the evaluation; a completely agentic tool would create a similar effect. Future work should investigate the impacts of agentic coding tools on learning outcomes and skill development.
- **Task Length:** Ideally, skill formation takes place over months to years. We measured skill formation for a specific Python library over a one-hour period. Future work should study real-world skill development through longitudinal measurement of the impacts of AI adoption.
- **Participant Realism:** While participants in our study were professional or freelance programmers, there was not the same incentive to learn the library as if it were required for their actual job. Future studies should aim at studying the skill acquisition from novice workers within a

real company. • Prompting Skills: We collect self-reported familiarity with AI coding tools, but we do not actually measure differences in prompting techniques. An extension to our work would also involve testing the level of prompting fluency beyond self-report. • Evaluation Design: Our study measures skill formation through a comprehensive quiz. Other studies could use the completion of another task or design coding as alternative evaluation strategies. • Human Assistance: We do not include the counterfactual of how skill formation would be impacted by receiving assistance from humans. Since human assistance and feedback takes place in a diverse settings (e.g., classroom, pair programming, code review), future work can compare the effect of feedback from AI vs humans in all these settings on skill formation. For novice workers in software engineering or any other industry, our study can be viewed as a small piece of evidence toward the value of intentional skill development despite the ubiquity of AI tools. Our study demonstrates the benefits of deploying cognitive effort when encountering a learning opportunity to master a new tool even if barriers (e.g., errors) may be encountered in the process of mastery. Exerting cognitive effort can be assisted by AI; beyond the patterns we describe, major LLM services also provide learning modes (e.g., ChatGPT Study Mode, Claude Code Learning / Explanatory mode). Ultimately, to accommodate skill development in the presence of AI, there needs to be a more expansive view of the impacts of AI on workers. Participants in the new AI economy must care not only about productivity gains from AI but also the long-term sustainability of expertise development amid the proliferation of new AI tools.

--- END PAPER TEXT ---

How AI Impacts Skill Formation

Method: Claim Evidence Extraction

Prompt B:

You are extracting claims and evidence from a research paper about AI-assisted software engineering.

Produce a table with exactly five rows and the following columns:

Claim	Evidence Snippet	Citation (source_id, chunk_id)

Constraints / Guardrails

Each claim must be directly supported by the provided evidence snippet.

Evidence snippets must be copied verbatim or closely paraphrased from the text.

Citations must use the format: (source_id, chunk_id).

Do not invent claims, evidence, or citations.

If the paper does not contain enough information to extract five well-supported claims, include fewer claims and explicitly state “Insufficient evidence for additional claims.”

Do not rely on external knowledge.

Paper: How AI Impacts Skill Formation

--- BEGIN PAPER TEXT ---

Abstract AI assistance produces significant productivity gains across professional domains, particularly for novice workers. Yet how this assistance affects the development of skills required to effectively supervise AI remains unclear. Novice workers who rely heavily on AI to complete unfamiliar tasks may compromise their own skill acquisition in the process. We conduct randomized experiments to study how developers gained mastery of a new asynchronous programming library with and without the assistance of AI. We find that AI use impairs conceptual understanding, code reading, and debugging abilities, without delivering significant efficiency gains on average. Participants who fully delegated coding tasks showed some productivity improvements, but at the cost of learning the library. We identify six distinct AI interaction patterns, three of which involve cognitive engagement and preserve learning outcomes even when participants receive AI assistance. Our findings suggest that AI-enhanced productivity is not a shortcut to competence and AI assistance should be carefully adopted into workflows to preserve skill formation – particularly in safety-critical domains. Introduction As AI promises to be a catalyst for automation and productivity in a wide range of applications, from software engineering to entrepreneurship [Dell'Acqua et al., 2023, Peng et al., 2023, Cui et al., 2024, Otis et al., 2024, Brynjolfsson et al., 2025], the impacts of AI on the labor force are not yet fully understood. Although more workers rely on AI to improve their productivity, it is unclear

whether the use of AI assistance in the workplace might hinder core understanding of concepts or prevent the development of skills necessary to supervise automated tasks. Although most studies have focused on the end product of AI assistance (e.g., lines of code written, quality of ideas proposed), an equally important, if not more crucial question is how process of receiving AI assistance impacts workers. As humans rely on AI for skills such as brainstorming, writing, and general critical thinking, the development of these skills may be significantly altered depending on how AI assistance is used. Software engineering, in particular, has been identified as a profession in which AI tools can be readily applied and AI assistance significantly improves productivity in daily tasks [Peng et al., 2023, Cui et al., 2024]. Junior or novice workers, in particular, benefit most from AI assistance when writing code. In high-stakes applications, AI written code may be debugged and tested by humans before a piece of software is ready for deployment. This additional verification that enhances safety is only possible when human engineers themselves have the skills to understand code and identify errors. As AI development progresses, the problem of supervising more and more capable AI systems becomes more difficult if humans have weaker abilities to understand code [Bowman et al., 2022]. When complex software tasks require human-AI collaboration, humans still need to understand the basic concepts of code development even if their software skills are complementary to the strengths of AI [Wang et al., 2020]. The combination of persistent competency requirements in high-stakes settings and demonstrated productivity gains from AI assistance makes software engineering an ideal testbed for studying how AI affects skill formation. We investigate whether using and relying on AI affects the development of software engineering skills [Handa et al., 2025]. Based on the rapid adoption of AI for software engineering, we are motivated by the scenario of engineers acquiring new skills on the job. Although the use of AI tools may improve productivity for these engineers, would they also inhibit skill formation? More specifically, does an AI-assisted task completion workflow prevent engineers from gaining in-depth knowledge about the tools used to complete these tasks? We run randomized experiments that measure skill formation by asking participants to complete coding tasks with a new library that they have not used before. This represents one way in which engineers acquire and learn new skills, since new libraries are frequently introduced in languages such as Python. We then evaluate their competency with the new library. Our main research questions are (1) whether AI improves productivity for a coding task requiring new concepts and skills, and (2) whether this use of AI reduces the level of understanding of these new concepts and skills.

Methods 4.1 Task Selection: Learning Asynchronous Programming with Trio

We prototyped tasks for several different skills that junior software engineers may encounter on the job: from data analysis to plotting. We designed an experiment around the Python Trio library,¹ which is designed for asynchronous concurrency and input-output processing (I/O). This library is less well known than asyncio (according to the number of StackOverflow questions) and involves new concepts (e.g., structured concurrency) beyond just Python fluency. It is also explicitly designed to be easy to use – making it particularly suitable for a learning experiment. Learned in a professional setting when working with large-scale data or software systems. The tasks we created include problem descriptions, starter code, and brief descriptions of the Trio concepts required to complete the task. These tasks are designed to parallel the process of learning to use a new library or new software tool through a brief self-guided tutorial. For example, in software engineers' on-boarding materials, there is often a description of how to use an internal library

and small tasks to build skills with the new library. After several pilot studies, we used the first two tasks in our main study; each task took 10 - 20 minutes during initial testing. The first task is to write a timer that prints every passing second while other functions run. This task introduces the core concepts of nurseries, starting tasks, and running functions concurrently in Trio. The second task involves implementing a record retrieval function that can handle missing record errors in the Trio library. This task introduces concepts such as error handling and memory channels to store results. These two tasks are standalone; we provide sufficient instructions and usage examples so that participants can complete one task without the other. We used an online interview platform with an AI assistant chat interface (Figure 3) for our experiments.

Participants in the AI condition are prompted to use the AI assistant to help them complete the task. The base model used for this assistant is GPT-4o, and the model is prompted to be an intelligent coding assistant. The AI assistant has access to participants' current version of the code and can produce the full, correct code for both tasks directly when prompted.

4.2 Evaluation Design Based on a previous meta-analysis of evaluations in computer science education [Cheng et al., 2022], we identify four types of questions used to assess the mastery of coding skills. Returning to our initial motivation of developing and retaining the skills required for supervising automation, proficiency in some of these areas may be more important than others for the oversight of AI-generated code. The four types of questions we consider are the following.

- **Debugging** The ability to identify and diagnose errors in code. This skill is crucial for detecting when AI-generated code is incorrect and understanding why it fails.
- **Code Reading** The ability to read and comprehend what code does. This skill enables humans to understand and verify AI-written code before deployment.
- **Code Writing** The ability to write or pick the right way to write code. Low-level code writing, like remembering the syntax of functions, will be less important with further integration of AI coding tools than high-level system design.
- **Conceptual** The ability to understand the core principles behind tools and libraries. Conceptual understanding is critical to assess whether AI-generated code uses appropriate design patterns that adheres to how the library should be used.

The two tasks in our study cover 7 core concepts from the Trio library. We designed a quiz with debugging, code reading, and conceptual questions that cover these 7 concepts. We exclude code writing questions to reduce the impact of syntax errors in our evaluation; these errors can be easily corrected with an AI query or web search. We tested 5 versions (Table 2) of the quiz in user testing and preliminary studies based on item response theory. For example, we ensure that all questions are sufficiently correlated with the overall quiz score, that each question has an appropriate average score, and that the questions are split up such that there is no local item dependence between questions (i.e., participants could not infer the answers to a question by looking at other questions). The final evaluation we used contained 14 questions for a total of 27 points. We submitted the grading rubric for the quiz in our study pre-registration before running the experiment.

4.3 Study Design We use a between-subjects randomized experiment to test for the effects of using AI in the coding skill formation process. Each participant first completed a warm-up coding task on a coding platform, where they needed to add a border around a list of strings. This Python coding question takes an average of 4 minutes to complete among users of this coding platform. There are no asynchronous concepts in this coding question. No participants have access to AI while completing the warm-up stage. We use this stage to calibrate the Python familiarity of the participants and to help participants familiarize themselves

with the interface. The next stage is the Trio task stage, where participants have a maximum of 35 minutes to complete two coding tasks using Trio in the same coding platform. During this stage, participants in the AI assistance condition (treatment group) had access to coding help through a chat-based AI assistant (Figure 3). All participants are instructed to complete the task as fast as they could. After completing the Trio task, participants completed the evaluation stage where they take the quiz we described in the previous section and complete a survey that consists of demographic and experiential questions after the quiz. In our main study, 52 participants completed the task, 26 for each of the control and treatment groups. For all our pilot studies and the main study, we only recruited participants who self-reported having more than one year of Python experience, code in Python at least once a week, have tried AI coding assistance at least a few times, and have never used the Trio library before (Table 1). We use the coding platform to collect the keystrokes of the users as they code and the transcripts of their interaction with the AI coding assistant in the coding condition. We use Google Forms to collect survey responses from users both before the coding task and after the coding task. Together, these tasks take a maximum time of 1 hour and 15 minutes with an average duration of 58.5 minutes. Participants were recruited through a third party crowd-worker platform and paid a flat rate of 150 USD for the task. Results Figure 6 shows that while using AI to complete our coding task did not significantly improve task completion time, the level of skill formation gained by completing the task, measured by our quiz, is significantly reduced (Cohen $d=0.738$, $p=0.01$). There is a 4.15 point difference between the means of the treatment and control groups. For a 27-point quiz, this translates into a 17% score difference or 2 grade points. Controlling for warm-up task time as a covariate, the treatment effect remains significant (Cohen's $d=0.725$, $p=0.016$). Prior works have presented mixed results on whether AI helps or hinders coding productivity [Peng et al., 2023, Becker et al., 2025]; our study differs from prior results in that it is designed to study how AI affects skill formation while performing a task requiring new knowledge. While we do observe a slightly lower average completion time in the AI group among novice programmers, due to the small group size of the 1-3 year participant group ($n=4$), the difference in task time was not significant. 4 of the 26 participants in the control (No AI) group did not complete the second task within the 35-minute limit, while every participant in the AI condition completed the second task. Our results do not conclusively find a speed up or slow down using AI in this task. Across all levels of prior coding experience, users scored higher on average in the control (no AI) than in the treatment (AI assistance) group (Figure 7). This shows that our choice of tasks and task design did not critically hinge on the participants' experience level of the but presented new skills to be acquired for every experience group. Concept Group Analysis In exploratory data analysis (not pre-registered), the quiz score was decomposed into subareas and question types (Figure 8). Each question in the quiz belonged to exactly one task (e.g., Task 1 or Task 2) and exactly one question type (e.g., Conceptual, Debugging, or Code Reading). For both tasks, there is a gap between the quiz scores between the treatment and control groups. Among the different types of questions, the largest score gap occurs in the debugging questions and the smallest score gap in the code reading questions. This outcome is expected since treatment and control groups may have similar exposure to reading code through the task, but the control group with no access to AI assistance encountered more errors during the task and became more capable at debugging. Task Experience In further exploratory data analysis, we also find differences in the way

participants' experience of completing the study. The control group (No AI) reported higher self-reported learning (on a 7-point scale), while both groups reported high levels of enjoyment in completing the task (Figure 9). In terms of difficulty of the task, Figure 10 shows that although participants in the treatment group (AI Assistance) found the task easier than the control group, both groups found the post-task quiz similarly challenging. Discussion Our main finding is that using AI to complete tasks that require a new skill (i.e., knowledge of a new Python library) reduces skill formation. In a randomized controlled trial, participants were assigned to the treatment condition (using an AI assistant, web search, and instructions) or the control condition (completing tasks with web search and instructions alone). The erosion of conceptual understanding, code reading, and debugging skills that we measured among participants using AI assistance suggests that workers acquiring new skills should be mindful of their reliance on AI during the learning process. Among participants who use AI, we find a stark divide in skill formation outcomes between high-scoring interaction patterns (65%-86% quiz score) vs low-scoring interaction patterns (24%-39% quiz score). The high scorers only asked AI conceptual questions instead of code generation or asked for explanations to accompany generated code; these usage patterns demonstrate a high level of cognitive engagement. Contrary to our initial hypothesis, we did not observe a significant performance boost in task completion in our main study. While using AI improved the average completion time of the task, the improvement in efficiency was not significant in our study, despite the AI Assistant being able to generate the complete code solution when prompted. Our qualitative analysis reveals that our finding is largely due to the heterogeneity in how participants decide to use AI during the task. There is a group of participants who relied on AI to generate all the code and never asked conceptual questions or for explanations. This group finished much faster than the control group (19.5 minutes vs 23 minutes), but this group only accounted for around 20% of the participants in the treatment group. Other participants in the AI group who asked a large number of queries (e.g., 15 queries), spent a long time composing queries (e.g., 10 minutes), or asked for follow-up explanations, raised the average task completion time. These contrasting patterns of AI usage suggest that accomplishing a task with new knowledge or skills does not necessarily lead to the same productive gains as tasks that require only existing knowledge. Together, our results suggest that the aggressive incorporation of AI into the workplace can have negative impacts on the professional development workers if they do not remain cognitatively engaged. Given time constraints and organizational pressures, junior developers or other professionals may rely on AI to complete tasks as fast as possible at the cost of real skill development. Furthermore, we found that the biggest difference in test scores is between the debugging questions. This suggests that as companies transition to more AI code writing with human supervision, humans may not possess the necessary skills to validate and debug AI-written code if their skill formation was inhibited by using AI in the first place.

7.1 Future Work

Our work is a first step to understanding the impact of AI assistance on humans in the human-AI collaboration process. We hope that this work will motivate future work that addresses the following limitations:

- Task Selection: This study focuses on a single task using a chat-based interface. This should be a lower bound for cognitive offloading since agentic AI coding tools would require even less human participation. In our work, users who relied on AI without thinking performed the worst on the evaluation; a completely agentic tool would create a similar effect. Future work should investigate the impacts of agentic coding tools on learning outcomes

and skill development. • Task Length: Ideally, skill formation takes place over months to years. We measured skill formation for a specific Python library over a one-hour period. Future work should study real-world skill development through longitudinal measurement of the impacts of AI adoption. • Participant Realism: While participants in our study were professional or freelance programmers, there was not the same incentive to learn the library as if it were required for their actual job. Future studies should aim at studying the skill acquisition from novice workers within a real company. • Prompting Skills: We collect self-reported familiarity with AI coding tools, but we do not actually measure differences in prompting techniques. An extension to our work would also involve testing the level of prompting fluency beyond self-report. • Evaluation Design: Our study measures skill formation through a comprehensive quiz. Other studies could use the completion of another task or design coding as alternative evaluation strategies. • Human Assistance: We do not include the counterfactual of how skill formation would be impacted by receiving assistance from humans. Since human assistance and feedback takes place in a diverse settings (e.g., classroom, pair programming, code review), future work can compare the effect of feedback from AI vs humans in all these settings on skill formation. For novice workers in software engineering or any other industry, our study can be viewed as a small piece of evidence toward the value of intentional skill development despite the ubiquity of AI tools. Our study demonstrates the benefits of deploying cognitive effort when encountering a learning opportunity to master a new tool even if barriers (e.g., errors) may be encountered in the process of mastery. Exerting cognitive effort can be assisted by AI; beyond the patterns we describe, major LLM services also provide learning modes (e.g., ChatGPT Study Mode, Claude Code Learning / Explanatory mode). Ultimately, to accommodate skill development in the presence of AI, there needs to be a more expansive view of the impacts of AI on workers. Participants in the new AI economy must care not only about productivity gains from AI but also the long-term sustainability of expertise development amid the proliferation of new AI tools.

--- END PAPER TEXT ---

Paper 3 Prompts

Quick Copy and Paste for:

- [PT Prompt A](#)
- [PT Prompt B](#)
- [CEE Prompt A](#)
- [CEE Prompt B](#)

The Good and Bad of AI Tools in Novice Programming Education

Method: Paper Triage

Prompt A:

You are given an academic paper about AI tools in software engineering.

Summarize the paper using the following fields:

Contribution

Method

Data

Findings

Limitations

Paper: The Good and Bad of AI Tools in Novice Programming Education

--- BEGIN PAPER TEXT ---

Abstract

As AI coding tools become more prevalent in programming, it is essential to understand how they influence programming education. This study, conducted in a first-semester Introduction to Programming course, aimed to determine the positive and negative effects of these tools on students' learning experiences and their ability to develop essential programming skills. Using a mixed-methods approach, we collected data from 73 teams of engineering students over a 12-week period. Students completed surveys and reported on their AI tool usage. We analyzed this data quantitatively to identify trends in tool familiarity, usage, and student satisfaction. Additionally, qualitative analysis of student reports provided insights into the specific ways AI tools were used and their perceived benefits and drawbacks. The findings revealed a significant increase in AI tool familiarity (from 28% to 100%) and usage among students. Students' satisfaction with AI tools improved over time. The most prevalent tasks for which novice programmers used AI tools included creating comments (91.7%), identifying and correcting bugs (80.2%), and seeking information (68.5%), while other tasks were less common. While these tools offered benefits like assisting in learning and enhancing real-world relevance, they

also raised concerns about cheating, over-reliance on AI tools, and a limited understanding of core programming concepts.

Aims and Research Questions

As we see, the central question is that of finding the right balance between the different aspects of the use of the tools. To determine the most effective approach, we need a clear understanding of how these tools are actually used in practice, particularly: how students perceive them and how they apply them in real-life programming courses. Especially interesting is observing students' first encounters with these tools, such as during their initial programming course in the first semester of their studies.

In this study, we aim to analyze novice programmers' attitudes toward AI tools. We seek to understand the dynamics of AI tool usage and changes in behavior over time. Additionally, we want to uncover patterns of AI usage and explore students' sentiments about these tools.

Essentially, our goal is to identify the 'Good' and the 'Bad' aspects of AI tool usage in novice programming education. To achieve this, we have formulated the following research questions:

RQ1. How familiar are novice programmers with AI tools during their programming education?

RQ2. Does the usage of AI tools evolve over time in an introductory programming course?

RQ3. Are students satisfied with the results provided by AI tools, and does this satisfaction improve over time?

RQ4. What common tasks do novice programming teams use AI tools for, and how prevalent are these tasks among the teams?

RQ5. What are the common benefits and concerns students have about using AI tools in their studies?

Methodology

The primary aims of this study are to understand the dynamics of AI tools usage, uncover patterns in their application, and explore students' sentiments regarding these tools. Data from students' reports and questionnaires on AI tool usage during the Introduction to Programming course are analyzed in this paper. The course spans a period of 12 weeks, with an instructional format consisting of 2 h of lectures and 4 h of lab sessions each week. A pragmatic mixed-methods study was undertaken to explore potential global patterns based on quantitative data, enriched with qualitative data. Students completed surveys and reported on their AI tool usage. The analytical framework of the study is presented in Figure 1.

2.1. Participants

The participants in this study were first-semester, first-year undergraduate students, who were novice programmers enrolled in the Introduction to Programming course. A total of 73 teams, each comprising two students from the Faculty of Engineering at Ruppin Academic Center in Israel, took part in the study. Students joined a team of their choice and remained with the same team until the end of the semester. Among the participants, 53% were male (78) and 47% were female (68). The median age was 24, with the range between 18 and 29 years. Only teams that submitted all required data items were included in this report. The data from these teams were used for the majority of the analysis in this study.

2.2. Procedure and Data Analysis

All procedures performed in this study were in accordance with APA ethical guidelines and the ethical standards of the institutional research committee. Prior to conducting the research, the requirements of the institutional ethics committee were complied with. At the beginning of the

semester, before starting the experimental process, the students were informed about the aims and procedures of the research. The tasks the students would undertake during the experimental process were explained. Each participant was instructed on how to create a unique identifier using a combination of personal but non-identifiable information. This identifier was used in all surveys and bonus questions throughout the semester. Team members merged the two personal, unique identifiers together.

Before the course started, participants completed a pre-semester questionnaire that included questions on students' personal information and basic knowledge of AI. At the end of the course, participants completed a post-semester questionnaire that assessed their perceptions of their AI knowledge, as well as their views on the benefits and concerns related to using AI tools in their studies.

Believing in the importance of incorporating AI tools into classrooms from the very beginning, specific assignment questions were designed where students were tasked with using AI tools to generate or explain portions of code, as well as to explore or learn about particular modules, concepts, or functions. Since these tasks required students to have a basic understanding of AI and the ability to effectively interact with AI tools, several examples were provided to demonstrate how these tools could be used in the context of their assignments. Although no specific AI tool was mandated, students were encouraged to explore different options to find what best suited their needs. As the formulation of the question of the student to the AI tool (usually called the prompt) could be important, in addition to these examples, prompt-writing tips were shared to help students craft more effective queries, enabling them to maximize the potential of the AI tools they chose to work with.

Furthermore, on a weekly basis, students worked on home assignments. A 5-point bonus question related to the usage of AI tools and students' feelings about them was added to some assignments. A bonus question is an additional question or set of questions included in an assignment that offers students the opportunity to earn extra points beyond the regular scoring criteria. To enable continuous data collection throughout the semester and gain insights into how teams interacted with AI tools as they progressed through the course, weekly multipart bonus questions were included in each assignment for the teams to answer. Detailed, ongoing data was gathered, mapping the progression of AI tool integration into their work and providing a nuanced understanding of how their usage evolved.

The course spanned 12 weeks, with weekly assignments categorized into two groups: those requiring the use of AI tools (assignments 3, 7, and 10) and those that did not (assignments 2, 6, 9, and 11). Bonus questions were incorporated into assignments 2, 3, 6, 7, 9, and 11, each having a slightly different structure based on whether AI tools were used. A sample of the bonus question questionnaires can be found in Appendix A. No bonus questions were included in the remaining weeks.

first_pageDownload PDFsettingsOrder Article Reprints

Open AccessArticle

The Good and Bad of AI Tools in Novice Programming Education

by Rina Zviel-GirshinORCID

The Center for Research in Technological and Engineering Education, Faculty of Engineering, Ruppin Academic Center, Kfar Monash 4025000, Israel

Educ. Sci. 2024, 14(10), 1089; <https://doi.org/10.3390/educsci14101089>

Submission received: 26 August 2024 / Revised: 29 September 2024 / Accepted: 30 September 2024 / Published: 6 October 2024

(This article belongs to the Special Issue How Artificial Intelligence Can Enhance Education: Current Practices and Challenges)

Download keyboard_arrow_down Browse Figure Review Reports Versions Notes

Abstract

As AI coding tools become more prevalent in programming, it is essential to understand how they influence programming education. This study, conducted in a first-semester Introduction to Programming course, aimed to determine the positive and negative effects of these tools on students' learning experiences and their ability to develop essential programming skills. Using a mixed-methods approach, we collected data from 73 teams of engineering students over a 12-week period. Students completed surveys and reported on their AI tool usage. We analyzed this data quantitatively to identify trends in tool familiarity, usage, and student satisfaction.

Additionally, qualitative analysis of student reports provided insights into the specific ways AI tools were used and their perceived benefits and drawbacks. The findings revealed a significant increase in AI tool familiarity (from 28% to 100%) and usage among students. Students' satisfaction with AI tools improved over time. The most prevalent tasks for which novice programmers used AI tools included creating comments (91.7%), identifying and correcting bugs (80.2%), and seeking information (68.5%), while other tasks were less common. While these tools offered benefits like assisting in learning and enhancing real-world relevance, they also raised concerns about cheating, over-reliance on AI tools, and a limited understanding of core programming concepts.

Keywords: artificial intelligence (AI); AI in education; programing education; introduction to programming; AI coding tools

1. Introduction

The integration of Artificial Intelligence (AI) and AI-driven coding tools in programming education has opened new avenues for enhancing both the teaching and learning experience. These tools, while offering significant benefits, can also be a double-edged sword. Our study raises important questions about the AI tools' long-term impact on students' understanding and proficiency. The potential of AI tools to transform education is undeniable. They can automate routine coding tasks, provide instant feedback, improve debugging and troubleshooting, and offer personalized, student-centered learning experiences. However, their use also introduces challenges, particularly regarding the balance between automation and preserving essential programming skills [1,2,3,4,5].

AI in education is rapidly emerging as a key area of focus in educational technology, yet its broader pedagogical implications remain a subject of ongoing debate. The uncertainty about how educators can effectively harness AI to create meaningful learning experiences in higher education is a major concern [3,6,7]. Educators are tasked with the crucial role of preparing students for a future where AI will be ubiquitous, not only in technology-driven industries but across various sectors. To achieve this, educators must equip students with a comprehensive skill set that includes traditional programming principles and knowledge in addition to the ability to leverage cutting-edge AI tools.

Teaching students to use AI tools in programming education aligns with the current demands of the high-tech industry and enhances their readiness for the workplace. By training students to

effectively utilize AI tools, educators can improve their efficiency and productivity, allowing them to focus on more complex and creative aspects of software development [8,9]. These tools can automate tedious and repetitive tasks, thereby freeing up cognitive resources for higher-order problem-solving and innovation [10,11]. However, the integration of AI tools in programming education must be approached with caution. Over-reliance on AI-driven solutions can result in a superficial understanding of programming concepts, where students may become adept at using tools without fully grasping the underlying logic and principles that drive software development [12,13].

A balanced approach is essential, where AI tools are used to complement, rather than replace, traditional coding practices and teaching methods [4,14]. This balance ensures that students develop a deep understanding of programming fundamentals while also learning to use AI to enhance their work. The primary objective of the current study is to investigate how college students use these tools in a first-semester introductory programming course and to explore how educators can achieve this balance, ensuring that AI tools are used to augment learning without diminishing the importance of manual coding and critical thinking [15]. The study also seeks to address the potential risks associated with AI tools, such as the risk of fostering surface-level learning and the possible erosion of essential problem-solving skills.

Ultimately, while AI tools offer significant potential for transforming programming education, their integration must be thoughtfully managed to ensure that they contribute positively to the educational process. By striking the right balance, educators can harness the power of AI to create more dynamic and effective learning environments that prepare students for the challenges of the future.

1.1. Literature Review

Code-generation tools powered by large language models can correctly and reliably solve most of programming problems that are typical in introductory courses. It is therefore unsurprising that Finnie-Ansley et al. [10] showed that AI tools could solve programming assignments and exam problems that are typically given in CS1 (computer science) and CS2 more effectively than most students. The implications of this capability extend beyond mere performance; they challenge the traditional approaches to teaching introductory programming.

Lau and Guo [13], in their semi-structured interviews with introductory programming instructors, revealed a range of perspectives on the integration of AI tools into education. While short-term concerns focused heavily on the potential for cheating, long-term perspectives were more varied. Some educators expressed a desire to resist the use of AI tools in introductory programming courses, concerned that such tools might undermine the development of fundamental skills. On the other hand, others advocated for embracing AI, suggesting that these tools could be integrated into the curriculum to enhance learning and better prepare students for a future where AI is ubiquitous in software development.

The debate about how to manage the influence of AI tools in education is further complicated by the issues of academic integrity and the assessment of student learning. Bommasani et al. [6] emphasized the increasing difficulty educators face in determining the extent of a student's individual contribution. The challenge of preventing ineffective collaborations and detecting plagiarism has become more pronounced as AI tools become more capable. This raises significant questions about the validity of traditional assessment methods and the need for new

approaches that can accurately measure a student's understanding and effort in an environment where AI-generated code is easily accessible.

Becker et al. [1] argued in their position paper that AI-generated code presents both opportunities and challenges for students and educators alike. On the one hand, these tools could serve as powerful aids, helping students overcome obstacles in their coding tasks and providing instant feedback that can enhance learning. On the other hand, they pose significant risks, such as the potential for students to become overly reliant on AI-generated solutions, which may hinder the development of critical thinking and problem-solving skills that are central to programming education.

Denny et al. [3] further argued that generative AI brings both challenges and opportunities to computing education, calling for updated pedagogical strategies that focus on new skill sets. The ability of generative AI models to generate solutions to problems typical of introductory programming courses raises concerns about student overreliance and misuse. However, this also opens the door for reimagining the educational landscape, where the focus shifts from rote learning and simple problem-solving to more complex tasks that require higher-order thinking and a deeper understanding of underlying concepts.

In a study conducted by Yilmaz and Yilmaz [11] on undergraduate students, it was found that tools like ChatGPT could enhance students' programming self-efficacy and motivation. The study suggested that ChatGPT could be a valuable tool for teaching computational thinking, providing students with a supportive environment where they can explore programming concepts with instant feedback. This potential for AI tools to positively influence student learning underscores the need for educators to consider how these tools can be integrated into the curriculum in a way that maximizes their benefits while mitigating their risks.

Several studies showed that ChatGPT could provide users with explanations, examples, and guidance to help them understand complex concepts and technologies. It could be used to assist with debugging by analyzing data on the programming language, code structure, error messages, and code documentation, and even assist with code review [16,17,18,19].

Vukojičić's and Krstić's [20] study found that using ChatGPT as a programming assistant resulted in better outcomes than working without external assistance. Students who used ChatGPT demonstrated improved coding proficiency, wrote higher-quality explanations, and gained a deeper understanding of standard solution methods.

Zhai [21] examined ChatGPT from the perspective of educators, asserting that the AI tool holds significant potential for offering specific and personalized guidance to both teachers and student teachers. The study found ChatGPT highly effective in automating essential tasks such as assessment development, grading, learning guidance, and recommending learning materials. This automation can greatly reduce the time and effort required for creating educational materials, enabling educators to dedicate more time to direct teaching and student engagement. However, to fully leverage these benefits, educators must possess the necessary technological, content, and professional expertise to access and evaluate the quality and relevance of the materials provided by ChatGPT.

Malinka et al. [22] studied the impact of ChatGPT on higher education, focusing primarily on computer security-oriented specializations. The study demonstrated the effectiveness and usability of ChatGPT for handling programming assignments, completing exams, and writing term papers. Various levels of tool misuse were evaluated, ranging from using it as a consultant

to simply copying its outputs. The benefits of ChatGPT for computer science education were also highlighted.

Rudolph et al. [23] examined the relevance of ChatGPT in the context of higher education, with a particular focus on its impact on assessment, learning, and teaching. Following an overview of ChatGPT's functionality and a summary of its strengths and limitations, the study delved into the broader implications of this technology for the future of higher education. The potential of AI chatbots like ChatGPT to reshape learning, teaching, and assessment practices was explored. ChatGPT was positioned within the current landscape of AI in education research, with its applications for students, educators, and educational systems discussed. The analysis highlighted both the opportunities and challenges presented by AI integration in education. Practical recommendations were proposed for students, educators, and higher education institutions on how to effectively navigate and leverage these emerging technologies.

Finnie-Ansley et al. [10] likened the introduction of Codex into students' hands to giving a power tool to an amateur—a tool with the potential to either construct or destruct, depending on how it is used. While the presence of AI tools in educational settings presents clear threats to student learning and academic integrity, it also offers fantastic opportunities to refactor existing curricula. By leveraging these tools appropriately, educators can design learning experiences that not only impart essential programming skills but also prepare students for a future where collaboration with AI is an integral part of the software development process.

Traditional teaching methods often suffer from low student participation, lack of personalized instruction, and insufficient motivation [24,25]. Modern approaches in education enhance active learning approaches [26]. Traditional education often relies on theory-based materials, but the integration of ChatGPT can foster personalized learning experiences tailored to individual needs and preferences. Students can leverage ChatGPT to outsource certain knowledge tasks, allowing them to concentrate on 'hands-on' learning and gain practical experience in their chosen fields. Additionally, ChatGPT could also create adaptive learning environments that respond to individual learner progress and performance [14,27,28].

Constructivist theory emphasizes that students construct knowledge through their own experience and participation. Integrating AI tools into a constructivist framework can significantly enhance educational experiences by encouraging students to take charge of their educational journeys through real-time assistance like clarification of concepts, answering questions, and offering tailored guidance based on individual learning needs and styles. With AI tools, educators can create environments that emphasize active learning. Students can engage in inquiry-based activities, utilizing AI as a resource for exploration and problem-solving [29,30,31,32,33]. AI integration may transform educators from content providers to facilitators, emphasizing mentorship and the development of soft skills [4,5,23,27].

Cognitive load theory posits that learning is most effective when cognitive load is managed, helping students focus on essential concepts without being overwhelmed [34,35]. AI tools can reduce extraneous cognitive load by automating routine tasks, allowing students to concentrate on understanding core programming principles. This can be particularly useful for novice programmers who may struggle with complex concepts.

Mandai and co-authors [36], in their opinion paper, discussed the potential impacts of ChatGPT on higher education through the lens of educational theories by John Dewey's Reflective Thought and Action model and the revised Bloom's taxonomy. The analysis was based on a

review of existing literature on ChatGPT and educational theories. The key points mentioned positive expectations: ChatGPT could enhance personalized learning, shift education towards more practical, hands-on learning by reducing the need for memorizing information, and lead to assessment reforms. They also mentioned negative expectations, like over-reliance on ChatGPT, failing to acquire necessary skills, diminished student creativity and originality, and they raised concerns about the authenticity and accuracy of sources.

Lo [32], in a literature review, provided an analysis of the implications of ChatGPT in educational contexts. The review assessed ChatGPT's capabilities across various subject domains, its potential applications in education, and the challenges it poses. ChatGPT's performance varied significantly across different subjects, demonstrating outstanding results in critical thinking and economics but showing unsatisfactory performance in mathematics and medical education. Potential applications in education include serving as an assistant for instructors or as a virtual tutor for students. Several challenges were identified, such as accuracy issues, plagiarism concerns, and the need for updated institutional policies to address these issues. The conclusion emphasized that while ChatGPT holds significant promise for enhancing educational practices, careful consideration must be given to its limitations and the ethical implications of its use.

1.2. Aims and Research Questions

As we see, the central question is that of finding the right balance between the different aspects of the use of the tools. To determine the most effective approach, we need a clear understanding of how these tools are actually used in practice, particularly: how students perceive them and how they apply them in real-life programming courses. Especially interesting is observing students' first encounters with these tools, such as during their initial programming course in the first semester of their studies.

In this study, we aim to analyze novice programmers' attitudes toward AI tools. We seek to understand the dynamics of AI tool usage and changes in behavior over time. Additionally, we want to uncover patterns of AI usage and explore students' sentiments about these tools.

Essentially, our goal is to identify the 'Good' and the 'Bad' aspects of AI tool usage in novice programming education. To achieve this, we have formulated the following research questions:

RQ1. How familiar are novice programmers with AI tools during their programming education?

RQ2. Does the usage of AI tools evolve over time in an introductory programming course?

RQ3. Are students satisfied with the results provided by AI tools, and does this satisfaction improve over time?

RQ4. What common tasks do novice programming teams use AI tools for, and how prevalent are these tasks among the teams?

RQ5. What are the common benefits and concerns students have about using AI tools in their studies?

2. Methodology

The primary aims of this study are to understand the dynamics of AI tools usage, uncover patterns in their application, and explore students' sentiments regarding these tools. Data from students' reports and questionnaires on AI tool usage during the Introduction to Programming course are analyzed in this paper. The course spans a period of 12 weeks, with an instructional format consisting of 2 h of lectures and 4 h of lab sessions each week. A pragmatic mixed-methods study was undertaken to explore potential global patterns based on quantitative

data, enriched with qualitative data. Students completed surveys and reported on their AI tool usage. The analytical framework of the study is presented in Figure 1.

Education 14 01089 g001

Figure 1. Analytical framework of the study.

2.1. Participants

The participants in this study were first-semester, first-year undergraduate students, who were novice programmers enrolled in the Introduction to Programming course. A total of 73 teams, each comprising two students from the Faculty of Engineering at Ruppin Academic Center in Israel, took part in the study. Students joined a team of their choice and remained with the same team until the end of the semester. Among the participants, 53% were male (78) and 47% were female (68). The median age was 24, with the range between 18 and 29 years. Only teams that submitted all required data items were included in this report. The data from these teams were used for the majority of the analysis in this study.

2.2. Procedure and Data Analysis

All procedures performed in this study were in accordance with APA ethical guidelines and the ethical standards of the institutional research committee. Prior to conducting the research, the requirements of the institutional ethics committee were complied with. At the beginning of the semester, before starting the experimental process, the students were informed about the aims and procedures of the research. The tasks the students would undertake during the experimental process were explained. Each participant was instructed on how to create a unique identifier using a combination of personal but non-identifiable information. This identifier was used in all surveys and bonus questions throughout the semester. Team members merged the two personal, unique identifiers together.

Before the course started, participants completed a pre-semester questionnaire that included questions on students' personal information and basic knowledge of AI. At the end of the course, participants completed a post-semester questionnaire that assessed their perceptions of their AI knowledge, as well as their views on the benefits and concerns related to using AI tools in their studies.

Believing in the importance of incorporating AI tools into classrooms from the very beginning, specific assignment questions were designed where students were tasked with using AI tools to generate or explain portions of code, as well as to explore or learn about particular modules, concepts, or functions. Since these tasks required students to have a basic understanding of AI and the ability to effectively interact with AI tools, several examples were provided to demonstrate how these tools could be used in the context of their assignments. Although no specific AI tool was mandated, students were encouraged to explore different options to find what best suited their needs. As the formulation of the question of the student to the AI tool (usually called the prompt) could be important, in addition to these examples, prompt-writing tips were shared to help students craft more effective queries, enabling them to maximize the potential of the AI tools they chose to work with.

Furthermore, on a weekly basis, students worked on home assignments. A 5-point bonus question related to the usage of AI tools and students' feelings about them was added to some assignments. A bonus question is an additional question or set of questions included in an assignment that offers students the opportunity to earn extra points beyond the regular scoring criteria. To enable continuous data collection throughout the semester and gain insights into

how teams interacted with AI tools as they progressed through the course, weekly multipart bonus questions were included in each assignment for the teams to answer. Detailed, ongoing data was gathered, mapping the progression of AI tool integration into their work and providing a nuanced understanding of how their usage evolved.

The course spanned 12 weeks, with weekly assignments categorized into two groups: those requiring the use of AI tools (assignments 3, 7, and 10) and those that did not (assignments 2, 6, 9, and 11). Bonus questions were incorporated into assignments 2, 3, 6, 7, 9, and 11, each having a slightly different structure based on whether AI tools were used. A sample of the bonus question questionnaires can be found in Appendix A. No bonus questions were included in the remaining weeks.

2.3. Data Analysis

The impact of AI tools was measured using several indicators: familiarity with AI tools was assessed through pre- and post-semester questionnaires that captured students' self-reported familiarity levels; the frequency of AI tool usage in assignments was evaluated with a dichotomous (yes/no) statement, "I used AI tools during this assignment", focusing on weeks where usage was not explicitly required to identify trends in voluntary adoption; students' comfort levels with AI tools were tracked during weeks 3, 7, and 10 using a 5-point Likert scale, allowing for the assessment of how comfort evolved as they became more accustomed to the tools; satisfaction with AI tools was measured through survey questions asking students to rate their satisfaction with the results provided by the tools on a Likert scale, offering insights into their perceptions of the quality of outcomes from AI tool usage.

IBM SPSS Statistics 28 software was applied for quantitative data analysis. Several statistical tests were used in this study: the Wilcoxon signed ranks test for two related samples, the Friedman test for several related samples, and Cochran's Q test. Cronbach's alpha coefficients, showing the internal consistency reliability, were computed. The values were acceptable: 0.935 for feeling comfortable with usage of AI tools in the assignment on weeks 3, 7, 10; 0.924 for using AI tools during the assignment on weeks 2, 6, 9; 0.758 for being happy with the results provided by AI tools on weeks 3, 7, 10.

For the qualitative data, participants' responses to the open-ended questions included in the bonus tasks were analyzed using content analysis. Prior to analyzing the data, two sets of categories based on prior research and literature were designed. The first set of categories was designed for responses to the statement, "I used AI tools during this assignment for the following tasks", and the second set was for responses to the direction, "Describe the benefits and concerns about using AI tools in your studies, personally".

We initially employed a deductive coding approach to categorize the data based on these predefined categories. However, several new categories emerged during the analysis, which were subsequently added to the pre-existing codes. Thus, a hybrid coding approach was adopted: the deductive coding provided a structured framework, while the inductive coding allowed for the discovery of additional categories, adding depth and nuance to the findings.

3. Results

Based on the responses received, we analyzed the results to address our research questions. We did not specify the use of particular tools. The majority of students utilized tools designed to understand and generate human-quality text in response to a variety of prompts and questions.

Most of these tools are user-friendly, featuring simple interfaces that allow users to input prompts and receive responses easily.

3.1. Familiarity with AI Tools

To answer the first research question about the familiarity of novice programmers with AI tools during their programming education, the level of agreement of the students to the following statement, “I feel familiar with AI tools usage”, was analyzed at the beginning and at the end of the course. The descriptive frequency and percentage of responses can be found in Table 1. The analysis of survey responses indicated that, initially, only 28% of the teams reported feeling familiar with the usage of AI tools. However, by the end of the course, this familiarity had increased to 100%.

Wilcoxon signed ranks test for two related samples was used to compare the responses at the beginning and at the end of the semester to the statement, “I feel familiar with AI tools usage”, measured on a Likert 1–5 scale. This test revealed significant differences between the responses at the beginning and at the end: $Z = -6.085$, $p < 0.001$.

3.2. Dynamics of AI Tool Integration

To address the second research question—whether the usage of AI tools evolves over time in an introductory programming course—the level of agreement with the statement, “I feel comfortable using AI tools in this assignment”, was analyzed on a Likert scale from 1 to 5 during weeks 3, 7, and 10. During these weeks, the use of AI tools was required to complete the assignments. The descriptive frequencies and percentages of responses are presented in Table 2.

The Friedman test for several related samples was applied, comparing the responses to the item, “I feel comfortable with usage of AI tools in this assignment”, on weeks 3, 7, and 10. The differences in medians were significant: $\chi^2 (2) = 9.8$, $p = 0.007$.

Additionally, to answer the second research question, we analyzed the level of agreement to the dichotomous (yes, no) statement, “I used AI tools during this assignment”, on weeks 2, 6, 9 and 11. Those weeks did not require the use of AI tools. The descriptive frequency and percentage of responses to the statement can be found in Table 3.

Non-parametric Cohran’s Q test was used to compare the response to the item, “I used AI tools during this assignment”, on weeks 2, 6, 9, and 11 (yes, no). Cohran’s $Q = 34.839$, $p < 0.001$, meaning that there were significant differences on the response between weeks 2, 6, 9, and 11, indicating that students used more AI tools.

3.3. Student Satisfaction with AI Tools

To address the third research question, “Are students satisfied with the results provided by AI tools, and does this satisfaction improve over time?”, we analyzed students’ level of agreement with the statement, “I was happy with the results provided by AI tools”, using a Likert scale of 1 to 5 during weeks 3, 7, and 10. During these specific weeks, the use of AI tools was mandatory for completing an assignment. The descriptive frequency and percentage of responses can be found in Table 4.

The Friedman test for several related samples was applied comparing the responses to the item, “I was happy with the results provided by AI tools”, on weeks 3, 7, and 10. The differences in medians were significant: $\chi^2 (2) = 11.594$, $p = 0.003$, meaning that the answers were better. However, since students used the same language model, this result suggests that they improved their prompt-engineering skills over time.

3.4. Common AI Tool Tasks and Prevalence

To address the fourth research question, “What common tasks do novice programming teams use AI tools for, and how prevalent are these tasks among the teams?”, we analyzed the responses to the statement, “I used AI tools during this assignment for the following tasks”. Each team was allowed to provide as many responses as they wished. Given that the quality of the code and its accompanying description becomes more demanding toward the end of the course, we focused on the teams’ responses from weeks 9 and 11. We combined the responses from each team across these two weeks, and if the same category was mentioned more than once, it was counted only once.

Categories that emerged from the analysis of the responses in the qualitative stage of the research and the percentage of the teams that expressed them were as follows: creation of comments mentioned by 67 teams (91.7%), bugs identification and correction—59 teams (80.2%), information seeking—50 teams (68.5%), code comparisons—33 teams (45.21%), creating some part of the code—25 teams (34.2%), debugging assistance—25 teams (34.2%), code simplification—14 teams (19.2%), translation of exercise from Hebrew (the original language) to another language (for example, Arabic or Russian)—12 teams (16.4%), algorithm selection—11 teams (15.1%). Translation of exercise from the original language to another language was a category that was added as a result of the inductive coding approach.

3.5. Benefits and Concerns of AI Tool Usage

To address the fifth research question about the common benefits and concerns students have regarding the use of AI tools in their studies, responses to the open-ended direction, “Describe the benefits and concerns you personally have about using AI tools in your studies”, during the last week of the course were analyzed. Common benefit categories emerged from the analysis of the responses, and the percentage of the teams that expressed them were as follows: instant help or quick answers and explanations mentioned by 58 teams (79.4%), assistance with repetitive tasks (like writing comments)—55 teams (75.3%), immediate response/feedback—42 teams (57.5%), help with identification and correction of errors—40 teams (54.8%), access to diverse resources/providing additional examples—33 teams (45.21%), spelling correction—17 teams (23.3%), translation from one language to another—14 teams (19.2%). Once again, translation from one language to another was added during the inductive coding process.

Common concern categories that emerged from the analysis of the responses and the percentage of the teams that expressed them were as follows: inaccurate or misleading information/answers mentioned by 65 teams (89.0%), over-reliance on AI tools/becoming too dependent on AI tools—55 teams (75.3%), identification of advanced functions and structures that were not learned in class—33 teams (45.21%), incorrect explanations which appear believable to novices—27 teams (36.9%), low code quality—25 teams (34.2%), wasting too much time on unsuccessful bug identification and error fixing—12 teams (16.4%), inconsistency/producing different outputs, even when given the same prompt—10 teams (13.6%). This last category was added as a result of the inductive coding approach.

4 Discussion

4.2. The Good

The positive ('Good') aspects can be summarized as follows: AI tools significantly enhanced participants' learning experiences by assisting with information retrieval, bug identification, and writing comments. Participants felt that the use of AI tools brought real-world relevance to the

course, fostering essential literacies and skills that are crucial for understanding technology and preparing them for the future. Additionally, some participants noted that receiving feedback from AI tools when they were stuck boosted their self-efficacy and motivation to learn. This finding aligns with Yilmaz and Yilmaz [11], who demonstrated that ChatGPT enhances student programming self-efficacy and motivation. Similarly, Yin et al. [15] found that chatbot-based learning effectively increases college students' motivation in basic computer science courses. Another advantage ('Good') of incorporating AI tools (beyond just programming courses) is their role in shaping learners' mindsets and skills related to AI, fostering their understanding and application of these technologies. Usage of these tools will help students in their future endeavors to understand the principles of AI, experience AI achievements, and implement AI applications effectively.

4.3. The Bad

On the negative ('Bad') side, challenges emerged, including instances of cheating, where students relied on AI tools to generate entire solutions, particularly under time pressure towards the end of the semester or when facing difficulties with certain learning topics. The efficiency of AI tools also led to the over-automation of some programming tasks, such as generating comments, with 92% of teams fully outsourcing this task to AI. Additionally, there were occasional issues with the quality of produced code, unsuccessful bug identification and error fixing, and the use of incorrect or non-existent functions and packages. AI can generate inaccurate outputs, inaccurate code, or incorrect explanations which appear believable to novices.

The generation of solutions (creating some part of the code) was mentioned only by a third of the teams, possibly because students perceive it as a violation of academic integrity. Despite this, the literature indicates that a high percentage of computer science students engage in some form of plagiarism, with some studies reporting the percentage being nearly 80% [39,40]. Sheard et al. [40] argued that when students are given tasks with readily available solutions in textbooks or lecture notes, they may be tempted to take shortcuts, thus bypassing the intended learning experience. In contrast, Albluwi [41], in his systematic review, noted that research on the relationship between pressure in computing courses and plagiarism is limited and does not adequately reflect the significant impact this factor has on academic dishonesty.

The 'Bad' side of AI tool usage highlighted several significant concerns. One of the primary issues is the risk of students developing an excessive or even blind reliance on these tools, which can lead to a superficial understanding of programming concepts and a diminished ability to solve problems independently. This reliance also opens the door to potential misuse, where the ease of accessibility and inclusion of these tools can tempt students to shortcut their learning process rather than engaging deeply with the material. As Chen et al. [42] mentioned, the value of a tool depends on its use, and there is the potential for Codex to be used in ways that limit learning, or ways that make the work of educators difficult. Actually, the developers of Codex mentioned one such challenge: possible over-reliance on Codex by novice programmers.

Furthermore, the overuse of code generation tools was found to result in a limited grasp of fundamental programming principles and concepts. This not only undermines students' ability to write and understand code on their own but also makes them more vulnerable to technical

issues and algorithmic errors. Such errors often arise from the AI's misinterpretation of tasks or from poorly formulated queries, leading to incorrect or suboptimal solutions.

Moreover, instead of fostering a collaborative learning environment where students work alongside AI to enhance their understanding, the AI is often doing most of the work. This dynamic can erode the learning experience, as students become passive recipients rather than active participants in the problem-solving process. The result is a concerning shift away from critical engagement with programming tasks towards a more superficial reliance on AI-generated solutions.

6. Limitations

This study has some limitations. First, the experimental process was conducted with 73 teams, which introduces the possibility that one team member may have been more dominant in the decision-making process. As a result, the outcomes and responses may reflect the perspectives or actions of that single dominant member, rather than representing the collective input of the entire team. In addition, the sample consists of a specific group of students from one course; therefore, the results might not be representative of all student populations. In further research, the number of participants can be increased, students from additional courses or institutions can join this study, and the wider results can be compared to current ones.

Additionally, a limitation of this study was the lack of direct measures of critical thinking skills. Future research incorporating such measures would provide a more comprehensive understanding of the impact of AI tools on student development. This addition would allow for a more precise evaluation of how AI tools affect students' critical thinking abilities, rather than relying solely on indirect indicators or self-reported data.

Also, much of the data relies on self-reported measures, such as students' comfort levels or perceived improvements. This could introduce biases, as students might overestimate or underestimate their abilities or comfort with AI tools.

--- END PAPER TEXT ---

The Good and Bad of AI Tools in Novice Programming Education

Method: Paper Triage

Prompt B:

You are analyzing an academic paper about AI-assisted tools in software engineering.

Produce a structured summary using exactly the following five fields:

Contribution

Method

Data

Findings

Limitations

Constraints / Guardrails

Base all statements strictly on the content of the provided paper.

Do not infer or speculate beyond what is explicitly supported by the text.

If the paper does not clearly specify information for a field, write: "Not specified in the paper."

Be concise, factual, and neutral in tone.

Do not introduce external knowledge or assumptions.

Output only the five fields listed above.

Paper: The Good and Bad of AI Tools in Novice Programming Education

--- BEGIN PAPER TEXT ---

Abstract

As AI coding tools become more prevalent in programming, it is essential to understand how they influence programming education. This study, conducted in a first-semester Introduction to Programming course, aimed to determine the positive and negative effects of these tools on students' learning experiences and their ability to develop essential programming skills. Using a mixed-methods approach, we collected data from 73 teams of engineering students over a 12-week period. Students completed surveys and reported on their AI tool usage. We analyzed this data quantitatively to identify trends in tool familiarity, usage, and student satisfaction.

Additionally, qualitative analysis of student reports provided insights into the specific ways AI tools were used and their perceived benefits and drawbacks. The findings revealed a significant increase in AI tool familiarity (from 28% to 100%) and usage among students. Students' satisfaction with AI tools improved over time. The most prevalent tasks for which novice programmers used AI tools included creating comments (91.7%), identifying and correcting bugs (80.2%), and seeking information (68.5%), while other tasks were less common. While these tools offered benefits like assisting in learning and enhancing real-world relevance, they

also raised concerns about cheating, over-reliance on AI tools, and a limited understanding of core programming concepts.

Aims and Research Questions

As we see, the central question is that of finding the right balance between the different aspects of the use of the tools. To determine the most effective approach, we need a clear understanding of how these tools are actually used in practice, particularly: how students perceive them and how they apply them in real-life programming courses. Especially interesting is observing students' first encounters with these tools, such as during their initial programming course in the first semester of their studies.

In this study, we aim to analyze novice programmers' attitudes toward AI tools. We seek to understand the dynamics of AI tool usage and changes in behavior over time. Additionally, we want to uncover patterns of AI usage and explore students' sentiments about these tools.

Essentially, our goal is to identify the 'Good' and the 'Bad' aspects of AI tool usage in novice programming education. To achieve this, we have formulated the following research questions:

RQ1. How familiar are novice programmers with AI tools during their programming education?

RQ2. Does the usage of AI tools evolve over time in an introductory programming course?

RQ3. Are students satisfied with the results provided by AI tools, and does this satisfaction improve over time?

RQ4. What common tasks do novice programming teams use AI tools for, and how prevalent are these tasks among the teams?

RQ5. What are the common benefits and concerns students have about using AI tools in their studies?

Methodology

The primary aims of this study are to understand the dynamics of AI tools usage, uncover patterns in their application, and explore students' sentiments regarding these tools. Data from students' reports and questionnaires on AI tool usage during the Introduction to Programming course are analyzed in this paper. The course spans a period of 12 weeks, with an instructional format consisting of 2 h of lectures and 4 h of lab sessions each week. A pragmatic mixed-methods study was undertaken to explore potential global patterns based on quantitative data, enriched with qualitative data. Students completed surveys and reported on their AI tool usage. The analytical framework of the study is presented in Figure 1.

2.1. Participants

The participants in this study were first-semester, first-year undergraduate students, who were novice programmers enrolled in the Introduction to Programming course. A total of 73 teams, each comprising two students from the Faculty of Engineering at Ruppin Academic Center in Israel, took part in the study. Students joined a team of their choice and remained with the same team until the end of the semester. Among the participants, 53% were male (78) and 47% were female (68). The median age was 24, with the range between 18 and 29 years. Only teams that submitted all required data items were included in this report. The data from these teams were used for the majority of the analysis in this study.

2.2. Procedure and Data Analysis

All procedures performed in this study were in accordance with APA ethical guidelines and the ethical standards of the institutional research committee. Prior to conducting the research, the requirements of the institutional ethics committee were complied with. At the beginning of the

semester, before starting the experimental process, the students were informed about the aims and procedures of the research. The tasks the students would undertake during the experimental process were explained. Each participant was instructed on how to create a unique identifier using a combination of personal but non-identifiable information. This identifier was used in all surveys and bonus questions throughout the semester. Team members merged the two personal, unique identifiers together.

Before the course started, participants completed a pre-semester questionnaire that included questions on students' personal information and basic knowledge of AI. At the end of the course, participants completed a post-semester questionnaire that assessed their perceptions of their AI knowledge, as well as their views on the benefits and concerns related to using AI tools in their studies.

Believing in the importance of incorporating AI tools into classrooms from the very beginning, specific assignment questions were designed where students were tasked with using AI tools to generate or explain portions of code, as well as to explore or learn about particular modules, concepts, or functions. Since these tasks required students to have a basic understanding of AI and the ability to effectively interact with AI tools, several examples were provided to demonstrate how these tools could be used in the context of their assignments. Although no specific AI tool was mandated, students were encouraged to explore different options to find what best suited their needs. As the formulation of the question of the student to the AI tool (usually called the prompt) could be important, in addition to these examples, prompt-writing tips were shared to help students craft more effective queries, enabling them to maximize the potential of the AI tools they chose to work with.

Furthermore, on a weekly basis, students worked on home assignments. A 5-point bonus question related to the usage of AI tools and students' feelings about them was added to some assignments. A bonus question is an additional question or set of questions included in an assignment that offers students the opportunity to earn extra points beyond the regular scoring criteria. To enable continuous data collection throughout the semester and gain insights into how teams interacted with AI tools as they progressed through the course, weekly multipart bonus questions were included in each assignment for the teams to answer. Detailed, ongoing data was gathered, mapping the progression of AI tool integration into their work and providing a nuanced understanding of how their usage evolved.

The course spanned 12 weeks, with weekly assignments categorized into two groups: those requiring the use of AI tools (assignments 3, 7, and 10) and those that did not (assignments 2, 6, 9, and 11). Bonus questions were incorporated into assignments 2, 3, 6, 7, 9, and 11, each having a slightly different structure based on whether AI tools were used. A sample of the bonus question questionnaires can be found in Appendix A. No bonus questions were included in the remaining weeks.

first_pageDownload PDFsettingsOrder Article Reprints

Open AccessArticle

The Good and Bad of AI Tools in Novice Programming Education

by Rina Zviel-GirshinORCID

The Center for Research in Technological and Engineering Education, Faculty of Engineering, Ruppin Academic Center, Kfar Monash 4025000, Israel

Educ. Sci. 2024, 14(10), 1089; <https://doi.org/10.3390/educsci14101089>

Submission received: 26 August 2024 / Revised: 29 September 2024 / Accepted: 30 September 2024 / Published: 6 October 2024

(This article belongs to the Special Issue How Artificial Intelligence Can Enhance Education: Current Practices and Challenges)

Download keyboard_arrow_down Browse Figure Review Reports Versions Notes

Abstract

As AI coding tools become more prevalent in programming, it is essential to understand how they influence programming education. This study, conducted in a first-semester Introduction to Programming course, aimed to determine the positive and negative effects of these tools on students' learning experiences and their ability to develop essential programming skills. Using a mixed-methods approach, we collected data from 73 teams of engineering students over a 12-week period. Students completed surveys and reported on their AI tool usage. We analyzed this data quantitatively to identify trends in tool familiarity, usage, and student satisfaction.

Additionally, qualitative analysis of student reports provided insights into the specific ways AI tools were used and their perceived benefits and drawbacks. The findings revealed a significant increase in AI tool familiarity (from 28% to 100%) and usage among students. Students' satisfaction with AI tools improved over time. The most prevalent tasks for which novice programmers used AI tools included creating comments (91.7%), identifying and correcting bugs (80.2%), and seeking information (68.5%), while other tasks were less common. While these tools offered benefits like assisting in learning and enhancing real-world relevance, they also raised concerns about cheating, over-reliance on AI tools, and a limited understanding of core programming concepts.

Keywords: artificial intelligence (AI); AI in education; programing education; introduction to programming; AI coding tools

1. Introduction

The integration of Artificial Intelligence (AI) and AI-driven coding tools in programming education has opened new avenues for enhancing both the teaching and learning experience. These tools, while offering significant benefits, can also be a double-edged sword. Our study raises important questions about the AI tools' long-term impact on students' understanding and proficiency. The potential of AI tools to transform education is undeniable. They can automate routine coding tasks, provide instant feedback, improve debugging and troubleshooting, and offer personalized, student-centered learning experiences. However, their use also introduces challenges, particularly regarding the balance between automation and preserving essential programming skills [1,2,3,4,5].

AI in education is rapidly emerging as a key area of focus in educational technology, yet its broader pedagogical implications remain a subject of ongoing debate. The uncertainty about how educators can effectively harness AI to create meaningful learning experiences in higher education is a major concern [3,6,7]. Educators are tasked with the crucial role of preparing students for a future where AI will be ubiquitous, not only in technology-driven industries but across various sectors. To achieve this, educators must equip students with a comprehensive skill set that includes traditional programming principles and knowledge in addition to the ability to leverage cutting-edge AI tools.

Teaching students to use AI tools in programming education aligns with the current demands of the high-tech industry and enhances their readiness for the workplace. By training students to

effectively utilize AI tools, educators can improve their efficiency and productivity, allowing them to focus on more complex and creative aspects of software development [8,9]. These tools can automate tedious and repetitive tasks, thereby freeing up cognitive resources for higher-order problem-solving and innovation [10,11]. However, the integration of AI tools in programming education must be approached with caution. Over-reliance on AI-driven solutions can result in a superficial understanding of programming concepts, where students may become adept at using tools without fully grasping the underlying logic and principles that drive software development [12,13].

A balanced approach is essential, where AI tools are used to complement, rather than replace, traditional coding practices and teaching methods [4,14]. This balance ensures that students develop a deep understanding of programming fundamentals while also learning to use AI to enhance their work. The primary objective of the current study is to investigate how college students use these tools in a first-semester introductory programming course and to explore how educators can achieve this balance, ensuring that AI tools are used to augment learning without diminishing the importance of manual coding and critical thinking [15]. The study also seeks to address the potential risks associated with AI tools, such as the risk of fostering surface-level learning and the possible erosion of essential problem-solving skills.

Ultimately, while AI tools offer significant potential for transforming programming education, their integration must be thoughtfully managed to ensure that they contribute positively to the educational process. By striking the right balance, educators can harness the power of AI to create more dynamic and effective learning environments that prepare students for the challenges of the future.

1.1. Literature Review

Code-generation tools powered by large language models can correctly and reliably solve most of programming problems that are typical in introductory courses. It is therefore unsurprising that Finnie-Ansley et al. [10] showed that AI tools could solve programming assignments and exam problems that are typically given in CS1 (computer science) and CS2 more effectively than most students. The implications of this capability extend beyond mere performance; they challenge the traditional approaches to teaching introductory programming.

Lau and Guo [13], in their semi-structured interviews with introductory programming instructors, revealed a range of perspectives on the integration of AI tools into education. While short-term concerns focused heavily on the potential for cheating, long-term perspectives were more varied. Some educators expressed a desire to resist the use of AI tools in introductory programming courses, concerned that such tools might undermine the development of fundamental skills. On the other hand, others advocated for embracing AI, suggesting that these tools could be integrated into the curriculum to enhance learning and better prepare students for a future where AI is ubiquitous in software development.

The debate about how to manage the influence of AI tools in education is further complicated by the issues of academic integrity and the assessment of student learning. Bommasani et al. [6] emphasized the increasing difficulty educators face in determining the extent of a student's individual contribution. The challenge of preventing ineffective collaborations and detecting plagiarism has become more pronounced as AI tools become more capable. This raises significant questions about the validity of traditional assessment methods and the need for new

approaches that can accurately measure a student's understanding and effort in an environment where AI-generated code is easily accessible.

Becker et al. [1] argued in their position paper that AI-generated code presents both opportunities and challenges for students and educators alike. On the one hand, these tools could serve as powerful aids, helping students overcome obstacles in their coding tasks and providing instant feedback that can enhance learning. On the other hand, they pose significant risks, such as the potential for students to become overly reliant on AI-generated solutions, which may hinder the development of critical thinking and problem-solving skills that are central to programming education.

Denny et al. [3] further argued that generative AI brings both challenges and opportunities to computing education, calling for updated pedagogical strategies that focus on new skill sets. The ability of generative AI models to generate solutions to problems typical of introductory programming courses raises concerns about student overreliance and misuse. However, this also opens the door for reimagining the educational landscape, where the focus shifts from rote learning and simple problem-solving to more complex tasks that require higher-order thinking and a deeper understanding of underlying concepts.

In a study conducted by Yilmaz and Yilmaz [11] on undergraduate students, it was found that tools like ChatGPT could enhance students' programming self-efficacy and motivation. The study suggested that ChatGPT could be a valuable tool for teaching computational thinking, providing students with a supportive environment where they can explore programming concepts with instant feedback. This potential for AI tools to positively influence student learning underscores the need for educators to consider how these tools can be integrated into the curriculum in a way that maximizes their benefits while mitigating their risks.

Several studies showed that ChatGPT could provide users with explanations, examples, and guidance to help them understand complex concepts and technologies. It could be used to assist with debugging by analyzing data on the programming language, code structure, error messages, and code documentation, and even assist with code review [16,17,18,19].

Vukojičić's and Krstić's [20] study found that using ChatGPT as a programming assistant resulted in better outcomes than working without external assistance. Students who used ChatGPT demonstrated improved coding proficiency, wrote higher-quality explanations, and gained a deeper understanding of standard solution methods.

Zhai [21] examined ChatGPT from the perspective of educators, asserting that the AI tool holds significant potential for offering specific and personalized guidance to both teachers and student teachers. The study found ChatGPT highly effective in automating essential tasks such as assessment development, grading, learning guidance, and recommending learning materials. This automation can greatly reduce the time and effort required for creating educational materials, enabling educators to dedicate more time to direct teaching and student engagement. However, to fully leverage these benefits, educators must possess the necessary technological, content, and professional expertise to access and evaluate the quality and relevance of the materials provided by ChatGPT.

Malinka et al. [22] studied the impact of ChatGPT on higher education, focusing primarily on computer security-oriented specializations. The study demonstrated the effectiveness and usability of ChatGPT for handling programming assignments, completing exams, and writing term papers. Various levels of tool misuse were evaluated, ranging from using it as a consultant

to simply copying its outputs. The benefits of ChatGPT for computer science education were also highlighted.

Rudolph et al. [23] examined the relevance of ChatGPT in the context of higher education, with a particular focus on its impact on assessment, learning, and teaching. Following an overview of ChatGPT's functionality and a summary of its strengths and limitations, the study delved into the broader implications of this technology for the future of higher education. The potential of AI chatbots like ChatGPT to reshape learning, teaching, and assessment practices was explored. ChatGPT was positioned within the current landscape of AI in education research, with its applications for students, educators, and educational systems discussed. The analysis highlighted both the opportunities and challenges presented by AI integration in education. Practical recommendations were proposed for students, educators, and higher education institutions on how to effectively navigate and leverage these emerging technologies.

Finnie-Ansley et al. [10] likened the introduction of Codex into students' hands to giving a power tool to an amateur—a tool with the potential to either construct or destruct, depending on how it is used. While the presence of AI tools in educational settings presents clear threats to student learning and academic integrity, it also offers fantastic opportunities to refactor existing curricula. By leveraging these tools appropriately, educators can design learning experiences that not only impart essential programming skills but also prepare students for a future where collaboration with AI is an integral part of the software development process.

Traditional teaching methods often suffer from low student participation, lack of personalized instruction, and insufficient motivation [24,25]. Modern approaches in education enhance active learning approaches [26]. Traditional education often relies on theory-based materials, but the integration of ChatGPT can foster personalized learning experiences tailored to individual needs and preferences. Students can leverage ChatGPT to outsource certain knowledge tasks, allowing them to concentrate on 'hands-on' learning and gain practical experience in their chosen fields. Additionally, ChatGPT could also create adaptive learning environments that respond to individual learner progress and performance [14,27,28].

Constructivist theory emphasizes that students construct knowledge through their own experience and participation. Integrating AI tools into a constructivist framework can significantly enhance educational experiences by encouraging students to take charge of their educational journeys through real-time assistance like clarification of concepts, answering questions, and offering tailored guidance based on individual learning needs and styles. With AI tools, educators can create environments that emphasize active learning. Students can engage in inquiry-based activities, utilizing AI as a resource for exploration and problem-solving [29,30,31,32,33]. AI integration may transform educators from content providers to facilitators, emphasizing mentorship and the development of soft skills [4,5,23,27].

Cognitive load theory posits that learning is most effective when cognitive load is managed, helping students focus on essential concepts without being overwhelmed [34,35]. AI tools can reduce extraneous cognitive load by automating routine tasks, allowing students to concentrate on understanding core programming principles. This can be particularly useful for novice programmers who may struggle with complex concepts.

Mandai and co-authors [36], in their opinion paper, discussed the potential impacts of ChatGPT on higher education through the lens of educational theories by John Dewey's Reflective Thought and Action model and the revised Bloom's taxonomy. The analysis was based on a

review of existing literature on ChatGPT and educational theories. The key points mentioned positive expectations: ChatGPT could enhance personalized learning, shift education towards more practical, hands-on learning by reducing the need for memorizing information, and lead to assessment reforms. They also mentioned negative expectations, like over-reliance on ChatGPT, failing to acquire necessary skills, diminished student creativity and originality, and they raised concerns about the authenticity and accuracy of sources.

Lo [32], in a literature review, provided an analysis of the implications of ChatGPT in educational contexts. The review assessed ChatGPT's capabilities across various subject domains, its potential applications in education, and the challenges it poses. ChatGPT's performance varied significantly across different subjects, demonstrating outstanding results in critical thinking and economics but showing unsatisfactory performance in mathematics and medical education. Potential applications in education include serving as an assistant for instructors or as a virtual tutor for students. Several challenges were identified, such as accuracy issues, plagiarism concerns, and the need for updated institutional policies to address these issues. The conclusion emphasized that while ChatGPT holds significant promise for enhancing educational practices, careful consideration must be given to its limitations and the ethical implications of its use.

1.2. Aims and Research Questions

As we see, the central question is that of finding the right balance between the different aspects of the use of the tools. To determine the most effective approach, we need a clear understanding of how these tools are actually used in practice, particularly: how students perceive them and how they apply them in real-life programming courses. Especially interesting is observing students' first encounters with these tools, such as during their initial programming course in the first semester of their studies.

In this study, we aim to analyze novice programmers' attitudes toward AI tools. We seek to understand the dynamics of AI tool usage and changes in behavior over time. Additionally, we want to uncover patterns of AI usage and explore students' sentiments about these tools.

Essentially, our goal is to identify the 'Good' and the 'Bad' aspects of AI tool usage in novice programming education. To achieve this, we have formulated the following research questions:

RQ1. How familiar are novice programmers with AI tools during their programming education?

RQ2. Does the usage of AI tools evolve over time in an introductory programming course?

RQ3. Are students satisfied with the results provided by AI tools, and does this satisfaction improve over time?

RQ4. What common tasks do novice programming teams use AI tools for, and how prevalent are these tasks among the teams?

RQ5. What are the common benefits and concerns students have about using AI tools in their studies?

2. Methodology

The primary aims of this study are to understand the dynamics of AI tools usage, uncover patterns in their application, and explore students' sentiments regarding these tools. Data from students' reports and questionnaires on AI tool usage during the Introduction to Programming course are analyzed in this paper. The course spans a period of 12 weeks, with an instructional format consisting of 2 h of lectures and 4 h of lab sessions each week. A pragmatic mixed-methods study was undertaken to explore potential global patterns based on quantitative

data, enriched with qualitative data. Students completed surveys and reported on their AI tool usage. The analytical framework of the study is presented in Figure 1.

Education 14 01089 g001

Figure 1. Analytical framework of the study.

2.1. Participants

The participants in this study were first-semester, first-year undergraduate students, who were novice programmers enrolled in the Introduction to Programming course. A total of 73 teams, each comprising two students from the Faculty of Engineering at Ruppin Academic Center in Israel, took part in the study. Students joined a team of their choice and remained with the same team until the end of the semester. Among the participants, 53% were male (78) and 47% were female (68). The median age was 24, with the range between 18 and 29 years. Only teams that submitted all required data items were included in this report. The data from these teams were used for the majority of the analysis in this study.

2.2. Procedure and Data Analysis

All procedures performed in this study were in accordance with APA ethical guidelines and the ethical standards of the institutional research committee. Prior to conducting the research, the requirements of the institutional ethics committee were complied with. At the beginning of the semester, before starting the experimental process, the students were informed about the aims and procedures of the research. The tasks the students would undertake during the experimental process were explained. Each participant was instructed on how to create a unique identifier using a combination of personal but non-identifiable information. This identifier was used in all surveys and bonus questions throughout the semester. Team members merged the two personal, unique identifiers together.

Before the course started, participants completed a pre-semester questionnaire that included questions on students' personal information and basic knowledge of AI. At the end of the course, participants completed a post-semester questionnaire that assessed their perceptions of their AI knowledge, as well as their views on the benefits and concerns related to using AI tools in their studies.

Believing in the importance of incorporating AI tools into classrooms from the very beginning, specific assignment questions were designed where students were tasked with using AI tools to generate or explain portions of code, as well as to explore or learn about particular modules, concepts, or functions. Since these tasks required students to have a basic understanding of AI and the ability to effectively interact with AI tools, several examples were provided to demonstrate how these tools could be used in the context of their assignments. Although no specific AI tool was mandated, students were encouraged to explore different options to find what best suited their needs. As the formulation of the question of the student to the AI tool (usually called the prompt) could be important, in addition to these examples, prompt-writing tips were shared to help students craft more effective queries, enabling them to maximize the potential of the AI tools they chose to work with.

Furthermore, on a weekly basis, students worked on home assignments. A 5-point bonus question related to the usage of AI tools and students' feelings about them was added to some assignments. A bonus question is an additional question or set of questions included in an assignment that offers students the opportunity to earn extra points beyond the regular scoring criteria. To enable continuous data collection throughout the semester and gain insights into

how teams interacted with AI tools as they progressed through the course, weekly multipart bonus questions were included in each assignment for the teams to answer. Detailed, ongoing data was gathered, mapping the progression of AI tool integration into their work and providing a nuanced understanding of how their usage evolved.

The course spanned 12 weeks, with weekly assignments categorized into two groups: those requiring the use of AI tools (assignments 3, 7, and 10) and those that did not (assignments 2, 6, 9, and 11). Bonus questions were incorporated into assignments 2, 3, 6, 7, 9, and 11, each having a slightly different structure based on whether AI tools were used. A sample of the bonus question questionnaires can be found in Appendix A. No bonus questions were included in the remaining weeks.

2.3. Data Analysis

The impact of AI tools was measured using several indicators: familiarity with AI tools was assessed through pre- and post-semester questionnaires that captured students' self-reported familiarity levels; the frequency of AI tool usage in assignments was evaluated with a dichotomous (yes/no) statement, "I used AI tools during this assignment", focusing on weeks where usage was not explicitly required to identify trends in voluntary adoption; students' comfort levels with AI tools were tracked during weeks 3, 7, and 10 using a 5-point Likert scale, allowing for the assessment of how comfort evolved as they became more accustomed to the tools; satisfaction with AI tools was measured through survey questions asking students to rate their satisfaction with the results provided by the tools on a Likert scale, offering insights into their perceptions of the quality of outcomes from AI tool usage.

IBM SPSS Statistics 28 software was applied for quantitative data analysis. Several statistical tests were used in this study: the Wilcoxon signed ranks test for two related samples, the Friedman test for several related samples, and Cochran's Q test. Cronbach's alpha coefficients, showing the internal consistency reliability, were computed. The values were acceptable: 0.935 for feeling comfortable with usage of AI tools in the assignment on weeks 3, 7, 10; 0.924 for using AI tools during the assignment on weeks 2, 6, 9; 0.758 for being happy with the results provided by AI tools on weeks 3, 7, 10.

For the qualitative data, participants' responses to the open-ended questions included in the bonus tasks were analyzed using content analysis. Prior to analyzing the data, two sets of categories based on prior research and literature were designed. The first set of categories was designed for responses to the statement, "I used AI tools during this assignment for the following tasks", and the second set was for responses to the direction, "Describe the benefits and concerns about using AI tools in your studies, personally".

We initially employed a deductive coding approach to categorize the data based on these predefined categories. However, several new categories emerged during the analysis, which were subsequently added to the pre-existing codes. Thus, a hybrid coding approach was adopted: the deductive coding provided a structured framework, while the inductive coding allowed for the discovery of additional categories, adding depth and nuance to the findings.

3. Results

Based on the responses received, we analyzed the results to address our research questions. We did not specify the use of particular tools. The majority of students utilized tools designed to understand and generate human-quality text in response to a variety of prompts and questions.

Most of these tools are user-friendly, featuring simple interfaces that allow users to input prompts and receive responses easily.

3.1. Familiarity with AI Tools

To answer the first research question about the familiarity of novice programmers with AI tools during their programming education, the level of agreement of the students to the following statement, “I feel familiar with AI tools usage”, was analyzed at the beginning and at the end of the course. The descriptive frequency and percentage of responses can be found in Table 1. The analysis of survey responses indicated that, initially, only 28% of the teams reported feeling familiar with the usage of AI tools. However, by the end of the course, this familiarity had increased to 100%.

Wilcoxon signed ranks test for two related samples was used to compare the responses at the beginning and at the end of the semester to the statement, “I feel familiar with AI tools usage”, measured on a Likert 1–5 scale. This test revealed significant differences between the responses at the beginning and at the end: $Z = -6.085$, $p < 0.001$.

3.2. Dynamics of AI Tool Integration

To address the second research question—whether the usage of AI tools evolves over time in an introductory programming course—the level of agreement with the statement, “I feel comfortable using AI tools in this assignment”, was analyzed on a Likert scale from 1 to 5 during weeks 3, 7, and 10. During these weeks, the use of AI tools was required to complete the assignments. The descriptive frequencies and percentages of responses are presented in Table 2.

The Friedman test for several related samples was applied, comparing the responses to the item, “I feel comfortable with usage of AI tools in this assignment”, on weeks 3, 7, and 10. The differences in medians were significant: $\chi^2 (2) = 9.8$, $p = 0.007$.

Additionally, to answer the second research question, we analyzed the level of agreement to the dichotomous (yes, no) statement, “I used AI tools during this assignment”, on weeks 2, 6, 9 and 11. Those weeks did not require the use of AI tools. The descriptive frequency and percentage of responses to the statement can be found in Table 3.

Non-parametric Cohran’s Q test was used to compare the response to the item, “I used AI tools during this assignment”, on weeks 2, 6, 9, and 11 (yes, no). Cohran’s $Q = 34.839$, $p < 0.001$, meaning that there were significant differences on the response between weeks 2, 6, 9, and 11, indicating that students used more AI tools.

3.3. Student Satisfaction with AI Tools

To address the third research question, “Are students satisfied with the results provided by AI tools, and does this satisfaction improve over time?”, we analyzed students’ level of agreement with the statement, “I was happy with the results provided by AI tools”, using a Likert scale of 1 to 5 during weeks 3, 7, and 10. During these specific weeks, the use of AI tools was mandatory for completing an assignment. The descriptive frequency and percentage of responses can be found in Table 4.

The Friedman test for several related samples was applied comparing the responses to the item, “I was happy with the results provided by AI tools”, on weeks 3, 7, and 10. The differences in medians were significant: $\chi^2 (2) = 11.594$, $p = 0.003$, meaning that the answers were better. However, since students used the same language model, this result suggests that they improved their prompt-engineering skills over time.

3.4. Common AI Tool Tasks and Prevalence

To address the fourth research question, “What common tasks do novice programming teams use AI tools for, and how prevalent are these tasks among the teams?”, we analyzed the responses to the statement, “I used AI tools during this assignment for the following tasks”. Each team was allowed to provide as many responses as they wished. Given that the quality of the code and its accompanying description becomes more demanding toward the end of the course, we focused on the teams’ responses from weeks 9 and 11. We combined the responses from each team across these two weeks, and if the same category was mentioned more than once, it was counted only once.

Categories that emerged from the analysis of the responses in the qualitative stage of the research and the percentage of the teams that expressed them were as follows: creation of comments mentioned by 67 teams (91.7%), bugs identification and correction—59 teams (80.2%), information seeking—50 teams (68.5%), code comparisons—33 teams (45.21%), creating some part of the code—25 teams (34.2%), debugging assistance—25 teams (34.2%), code simplification—14 teams (19.2%), translation of exercise from Hebrew (the original language) to another language (for example, Arabic or Russian)—12 teams (16.4%), algorithm selection—11 teams (15.1%). Translation of exercise from the original language to another language was a category that was added as a result of the inductive coding approach.

3.5. Benefits and Concerns of AI Tool Usage

To address the fifth research question about the common benefits and concerns students have regarding the use of AI tools in their studies, responses to the open-ended direction, “Describe the benefits and concerns you personally have about using AI tools in your studies”, during the last week of the course were analyzed. Common benefit categories emerged from the analysis of the responses, and the percentage of the teams that expressed them were as follows: instant help or quick answers and explanations mentioned by 58 teams (79.4%), assistance with repetitive tasks (like writing comments)—55 teams (75.3%), immediate response/feedback—42 teams (57.5%), help with identification and correction of errors—40 teams (54.8%), access to diverse resources/providing additional examples—33 teams (45.21%), spelling correction—17 teams (23.3%), translation from one language to another—14 teams (19.2%). Once again, translation from one language to another was added during the inductive coding process.

Common concern categories that emerged from the analysis of the responses and the percentage of the teams that expressed them were as follows: inaccurate or misleading information/answers mentioned by 65 teams (89.0%), over-reliance on AI tools/becoming too dependent on AI tools—55 teams (75.3%), identification of advanced functions and structures that were not learned in class—33 teams (45.21%), incorrect explanations which appear believable to novices—27 teams (36.9%), low code quality—25 teams (34.2%), wasting too much time on unsuccessful bug identification and error fixing—12 teams (16.4%), inconsistency/producing different outputs, even when given the same prompt—10 teams (13.6%). This last category was added as a result of the inductive coding approach.

4 Discussion

4.2. The Good

The positive ('Good') aspects can be summarized as follows: AI tools significantly enhanced participants' learning experiences by assisting with information retrieval, bug identification, and writing comments. Participants felt that the use of AI tools brought real-world relevance to the

course, fostering essential literacies and skills that are crucial for understanding technology and preparing them for the future. Additionally, some participants noted that receiving feedback from AI tools when they were stuck boosted their self-efficacy and motivation to learn. This finding aligns with Yilmaz and Yilmaz [11], who demonstrated that ChatGPT enhances student programming self-efficacy and motivation. Similarly, Yin et al. [15] found that chatbot-based learning effectively increases college students' motivation in basic computer science courses. Another advantage ('Good') of incorporating AI tools (beyond just programming courses) is their role in shaping learners' mindsets and skills related to AI, fostering their understanding and application of these technologies. Usage of these tools will help students in their future endeavors to understand the principles of AI, experience AI achievements, and implement AI applications effectively.

4.3. The Bad

On the negative ('Bad') side, challenges emerged, including instances of cheating, where students relied on AI tools to generate entire solutions, particularly under time pressure towards the end of the semester or when facing difficulties with certain learning topics. The efficiency of AI tools also led to the over-automation of some programming tasks, such as generating comments, with 92% of teams fully outsourcing this task to AI. Additionally, there were occasional issues with the quality of produced code, unsuccessful bug identification and error fixing, and the use of incorrect or non-existent functions and packages. AI can generate inaccurate outputs, inaccurate code, or incorrect explanations which appear believable to novices.

The generation of solutions (creating some part of the code) was mentioned only by a third of the teams, possibly because students perceive it as a violation of academic integrity. Despite this, the literature indicates that a high percentage of computer science students engage in some form of plagiarism, with some studies reporting the percentage being nearly 80% [39,40]. Sheard et al. [40] argued that when students are given tasks with readily available solutions in textbooks or lecture notes, they may be tempted to take shortcuts, thus bypassing the intended learning experience. In contrast, Albluwi [41], in his systematic review, noted that research on the relationship between pressure in computing courses and plagiarism is limited and does not adequately reflect the significant impact this factor has on academic dishonesty.

The 'Bad' side of AI tool usage highlighted several significant concerns. One of the primary issues is the risk of students developing an excessive or even blind reliance on these tools, which can lead to a superficial understanding of programming concepts and a diminished ability to solve problems independently. This reliance also opens the door to potential misuse, where the ease of accessibility and inclusion of these tools can tempt students to shortcut their learning process rather than engaging deeply with the material. As Chen et al. [42] mentioned, the value of a tool depends on its use, and there is the potential for Codex to be used in ways that limit learning, or ways that make the work of educators difficult. Actually, the developers of Codex mentioned one such challenge: possible over-reliance on Codex by novice programmers.

Furthermore, the overuse of code generation tools was found to result in a limited grasp of fundamental programming principles and concepts. This not only undermines students' ability to write and understand code on their own but also makes them more vulnerable to technical

issues and algorithmic errors. Such errors often arise from the AI's misinterpretation of tasks or from poorly formulated queries, leading to incorrect or suboptimal solutions.

Moreover, instead of fostering a collaborative learning environment where students work alongside AI to enhance their understanding, the AI is often doing most of the work. This dynamic can erode the learning experience, as students become passive recipients rather than active participants in the problem-solving process. The result is a concerning shift away from critical engagement with programming tasks towards a more superficial reliance on AI-generated solutions.

6. Limitations

This study has some limitations. First, the experimental process was conducted with 73 teams, which introduces the possibility that one team member may have been more dominant in the decision-making process. As a result, the outcomes and responses may reflect the perspectives or actions of that single dominant member, rather than representing the collective input of the entire team. In addition, the sample consists of a specific group of students from one course; therefore, the results might not be representative of all student populations. In further research, the number of participants can be increased, students from additional courses or institutions can join this study, and the wider results can be compared to current ones.

Additionally, a limitation of this study was the lack of direct measures of critical thinking skills. Future research incorporating such measures would provide a more comprehensive understanding of the impact of AI tools on student development. This addition would allow for a more precise evaluation of how AI tools affect students' critical thinking abilities, rather than relying solely on indirect indicators or self-reported data.

Also, much of the data relies on self-reported measures, such as students' comfort levels or perceived improvements. This could introduce biases, as students might overestimate or underestimate their abilities or comfort with AI tools.

--- END PAPER TEXT ---

The Good and Bad of AI Tools in Novice Programming Education

Method: Claim Evidence Extraction

Prompt A:

From the provided paper, extract five key claims about the impact of AI tools on software engineering.

For each claim, include:

The claim

A supporting quote or snippet from the text

A citation

Present your answer in a table.

Paper: The Good and Bad of AI Tools in Novice Programming Education

--- BEGIN PAPER TEXT ---

Abstract

As AI coding tools become more prevalent in programming, it is essential to understand how they influence programming education. This study, conducted in a first-semester Introduction to Programming course, aimed to determine the positive and negative effects of these tools on students' learning experiences and their ability to develop essential programming skills. Using a mixed-methods approach, we collected data from 73 teams of engineering students over a 12-week period. Students completed surveys and reported on their AI tool usage. We analyzed this data quantitatively to identify trends in tool familiarity, usage, and student satisfaction.

Additionally, qualitative analysis of student reports provided insights into the specific ways AI tools were used and their perceived benefits and drawbacks. The findings revealed a significant increase in AI tool familiarity (from 28% to 100%) and usage among students. Students' satisfaction with AI tools improved over time. The most prevalent tasks for which novice programmers used AI tools included creating comments (91.7%), identifying and correcting bugs (80.2%), and seeking information (68.5%), while other tasks were less common. While these tools offered benefits like assisting in learning and enhancing real-world relevance, they also raised concerns about cheating, over-reliance on AI tools, and a limited understanding of core programming concepts.

Aims and Research Questions

As we see, the central question is that of finding the right balance between the different aspects of the use of the tools. To determine the most effective approach, we need a clear understanding of how these tools are actually used in practice, particularly: how students perceive them and how they apply them in real-life programming courses. Especially interesting

is observing students' first encounters with these tools, such as during their initial programming course in the first semester of their studies.

In this study, we aim to analyze novice programmers' attitudes toward AI tools. We seek to understand the dynamics of AI tool usage and changes in behavior over time. Additionally, we want to uncover patterns of AI usage and explore students' sentiments about these tools.

Essentially, our goal is to identify the 'Good' and the 'Bad' aspects of AI tool usage in novice programming education. To achieve this, we have formulated the following research questions:

RQ1. How familiar are novice programmers with AI tools during their programming education?

RQ2. Does the usage of AI tools evolve over time in an introductory programming course?

RQ3. Are students satisfied with the results provided by AI tools, and does this satisfaction improve over time?

RQ4. What common tasks do novice programming teams use AI tools for, and how prevalent are these tasks among the teams?

RQ5. What are the common benefits and concerns students have about using AI tools in their studies?

Methodology

The primary aims of this study are to understand the dynamics of AI tools usage, uncover patterns in their application, and explore students' sentiments regarding these tools. Data from students' reports and questionnaires on AI tool usage during the Introduction to Programming course are analyzed in this paper. The course spans a period of 12 weeks, with an instructional format consisting of 2 h of lectures and 4 h of lab sessions each week. A pragmatic mixed-methods study was undertaken to explore potential global patterns based on quantitative data, enriched with qualitative data. Students completed surveys and reported on their AI tool usage. The analytical framework of the study is presented in Figure 1.

2.1. Participants

The participants in this study were first-semester, first-year undergraduate students, who were novice programmers enrolled in the Introduction to Programming course. A total of 73 teams, each comprising two students from the Faculty of Engineering at Ruppin Academic Center in Israel, took part in the study. Students joined a team of their choice and remained with the same team until the end of the semester. Among the participants, 53% were male (78) and 47% were female (68). The median age was 24, with the range between 18 and 29 years. Only teams that submitted all required data items were included in this report. The data from these teams were used for the majority of the analysis in this study.

2.2. Procedure and Data Analysis

All procedures performed in this study were in accordance with APA ethical guidelines and the ethical standards of the institutional research committee. Prior to conducting the research, the requirements of the institutional ethics committee were complied with. At the beginning of the semester, before starting the experimental process, the students were informed about the aims and procedures of the research. The tasks the students would undertake during the experimental process were explained. Each participant was instructed on how to create a unique identifier using a combination of personal but non-identifiable information. This identifier was used in all surveys and bonus questions throughout the semester. Team members merged the two personal, unique identifiers together.

Before the course started, participants completed a pre-semester questionnaire that included questions on students' personal information and basic knowledge of AI. At the end of the course, participants completed a post-semester questionnaire that assessed their perceptions of their AI knowledge, as well as their views on the benefits and concerns related to using AI tools in their studies.

Believing in the importance of incorporating AI tools into classrooms from the very beginning, specific assignment questions were designed where students were tasked with using AI tools to generate or explain portions of code, as well as to explore or learn about particular modules, concepts, or functions. Since these tasks required students to have a basic understanding of AI and the ability to effectively interact with AI tools, several examples were provided to demonstrate how these tools could be used in the context of their assignments. Although no specific AI tool was mandated, students were encouraged to explore different options to find what best suited their needs. As the formulation of the question of the student to the AI tool (usually called the prompt) could be important, in addition to these examples, prompt-writing tips were shared to help students craft more effective queries, enabling them to maximize the potential of the AI tools they chose to work with.

Furthermore, on a weekly basis, students worked on home assignments. A 5-point bonus question related to the usage of AI tools and students' feelings about them was added to some assignments. A bonus question is an additional question or set of questions included in an assignment that offers students the opportunity to earn extra points beyond the regular scoring criteria. To enable continuous data collection throughout the semester and gain insights into how teams interacted with AI tools as they progressed through the course, weekly multipart bonus questions were included in each assignment for the teams to answer. Detailed, ongoing data was gathered, mapping the progression of AI tool integration into their work and providing a nuanced understanding of how their usage evolved.

The course spanned 12 weeks, with weekly assignments categorized into two groups: those requiring the use of AI tools (assignments 3, 7, and 10) and those that did not (assignments 2, 6, 9, and 11). Bonus questions were incorporated into assignments 2, 3, 6, 7, 9, and 11, each having a slightly different structure based on whether AI tools were used. A sample of the bonus question questionnaires can be found in Appendix A. No bonus questions were included in the remaining weeks.

first_pageDownload PDFsettingsOrder Article Reprints

Open AccessArticle

The Good and Bad of AI Tools in Novice Programming Education

by Rina Zviel-GirshinORCID

The Center for Research in Technological and Engineering Education, Faculty of Engineering, Ruppin Academic Center, Kfar Monash 4025000, Israel

Educ. Sci. 2024, 14(10), 1089; <https://doi.org/10.3390/educsci14101089>

Submission received: 26 August 2024 / Revised: 29 September 2024 / Accepted: 30 September 2024 / Published: 6 October 2024

(This article belongs to the Special Issue How Artificial Intelligence Can Enhance Education: Current Practices and Challenges)

Downloadkeyboard_arrow_down Browse Figure Review Reports Versions Notes

Abstract

As AI coding tools become more prevalent in programming, it is essential to understand how they influence programming education. This study, conducted in a first-semester Introduction to Programming course, aimed to determine the positive and negative effects of these tools on students' learning experiences and their ability to develop essential programming skills. Using a mixed-methods approach, we collected data from 73 teams of engineering students over a 12-week period. Students completed surveys and reported on their AI tool usage. We analyzed this data quantitatively to identify trends in tool familiarity, usage, and student satisfaction. Additionally, qualitative analysis of student reports provided insights into the specific ways AI tools were used and their perceived benefits and drawbacks. The findings revealed a significant increase in AI tool familiarity (from 28% to 100%) and usage among students. Students' satisfaction with AI tools improved over time. The most prevalent tasks for which novice programmers used AI tools included creating comments (91.7%), identifying and correcting bugs (80.2%), and seeking information (68.5%), while other tasks were less common. While these tools offered benefits like assisting in learning and enhancing real-world relevance, they also raised concerns about cheating, over-reliance on AI tools, and a limited understanding of core programming concepts.

Keywords: artificial intelligence (AI); AI in education; programing education; introduction to programming; AI coding tools

1. Introduction

The integration of Artificial Intelligence (AI) and AI-driven coding tools in programming education has opened new avenues for enhancing both the teaching and learning experience. These tools, while offering significant benefits, can also be a double-edged sword. Our study raises important questions about the AI tools' long-term impact on students' understanding and proficiency. The potential of AI tools to transform education is undeniable. They can automate routine coding tasks, provide instant feedback, improve debugging and troubleshooting, and offer personalized, student-centered learning experiences. However, their use also introduces challenges, particularly regarding the balance between automation and preserving essential programming skills [1,2,3,4,5].

AI in education is rapidly emerging as a key area of focus in educational technology, yet its broader pedagogical implications remain a subject of ongoing debate. The uncertainty about how educators can effectively harness AI to create meaningful learning experiences in higher education is a major concern [3,6,7]. Educators are tasked with the crucial role of preparing students for a future where AI will be ubiquitous, not only in technology-driven industries but across various sectors. To achieve this, educators must equip students with a comprehensive skill set that includes traditional programming principles and knowledge in addition to the ability to leverage cutting-edge AI tools.

Teaching students to use AI tools in programming education aligns with the current demands of the high-tech industry and enhances their readiness for the workplace. By training students to effectively utilize AI tools, educators can improve their efficiency and productivity, allowing them to focus on more complex and creative aspects of software development [8,9]. These tools can automate tedious and repetitive tasks, thereby freeing up cognitive resources for higher-order problem-solving and innovation [10,11]. However, the integration of AI tools in programming education must be approached with caution. Over-reliance on AI-driven solutions can result in a superficial understanding of programming concepts, where students may become adept at

using tools without fully grasping the underlying logic and principles that drive software development [12,13].

A balanced approach is essential, where AI tools are used to complement, rather than replace, traditional coding practices and teaching methods [4,14]. This balance ensures that students develop a deep understanding of programming fundamentals while also learning to use AI to enhance their work. The primary objective of the current study is to investigate how college students use these tools in a first-semester introductory programming course and to explore how educators can achieve this balance, ensuring that AI tools are used to augment learning without diminishing the importance of manual coding and critical thinking [15]. The study also seeks to address the potential risks associated with AI tools, such as the risk of fostering surface-level learning and the possible erosion of essential problem-solving skills.

Ultimately, while AI tools offer significant potential for transforming programming education, their integration must be thoughtfully managed to ensure that they contribute positively to the educational process. By striking the right balance, educators can harness the power of AI to create more dynamic and effective learning environments that prepare students for the challenges of the future.

1.1. Literature Review

Code-generation tools powered by large language models can correctly and reliably solve most of programming problems that are typical in introductory courses. It is therefore unsurprising that Finnie-Ansley et al. [10] showed that AI tools could solve programming assignments and exam problems that are typically given in CS1 (computer science) and CS2 more effectively than most students. The implications of this capability extend beyond mere performance; they challenge the traditional approaches to teaching introductory programming.

Lau and Guo [13], in their semi-structured interviews with introductory programming instructors, revealed a range of perspectives on the integration of AI tools into education. While short-term concerns focused heavily on the potential for cheating, long-term perspectives were more varied. Some educators expressed a desire to resist the use of AI tools in introductory programming courses, concerned that such tools might undermine the development of fundamental skills. On the other hand, others advocated for embracing AI, suggesting that these tools could be integrated into the curriculum to enhance learning and better prepare students for a future where AI is ubiquitous in software development.

The debate about how to manage the influence of AI tools in education is further complicated by the issues of academic integrity and the assessment of student learning. Bommasani et al. [6] emphasized the increasing difficulty educators face in determining the extent of a student's individual contribution. The challenge of preventing ineffective collaborations and detecting plagiarism has become more pronounced as AI tools become more capable. This raises significant questions about the validity of traditional assessment methods and the need for new approaches that can accurately measure a student's understanding and effort in an environment where AI-generated code is easily accessible.

Becker et al. [1] argued in their position paper that AI-generated code presents both opportunities and challenges for students and educators alike. On the one hand, these tools could serve as powerful aids, helping students overcome obstacles in their coding tasks and providing instant feedback that can enhance learning. On the other hand, they pose significant risks, such as the potential for students to become overly reliant on AI-generated solutions,

which may hinder the development of critical thinking and problem-solving skills that are central to programming education.

Denny et al. [3] further argued that generative AI brings both challenges and opportunities to computing education, calling for updated pedagogical strategies that focus on new skill sets. The ability of generative AI models to generate solutions to problems typical of introductory programming courses raises concerns about student overreliance and misuse. However, this also opens the door for reimagining the educational landscape, where the focus shifts from rote learning and simple problem-solving to more complex tasks that require higher-order thinking and a deeper understanding of underlying concepts.

In a study conducted by Yilmaz and Yilmaz [11] on undergraduate students, it was found that tools like ChatGPT could enhance students' programming self-efficacy and motivation. The study suggested that ChatGPT could be a valuable tool for teaching computational thinking, providing students with a supportive environment where they can explore programming concepts with instant feedback. This potential for AI tools to positively influence student learning underscores the need for educators to consider how these tools can be integrated into the curriculum in a way that maximizes their benefits while mitigating their risks.

Several studies showed that ChatGPT could provide users with explanations, examples, and guidance to help them understand complex concepts and technologies. It could be used to assist with debugging by analyzing data on the programming language, code structure, error messages, and code documentation, and even assist with code review [16,17,18,19].

Vukojičić's and Krstić's [20] study found that using ChatGPT as a programming assistant resulted in better outcomes than working without external assistance. Students who used ChatGPT demonstrated improved coding proficiency, wrote higher-quality explanations, and gained a deeper understanding of standard solution methods.

Zhai [21] examined ChatGPT from the perspective of educators, asserting that the AI tool holds significant potential for offering specific and personalized guidance to both teachers and student teachers. The study found ChatGPT highly effective in automating essential tasks such as assessment development, grading, learning guidance, and recommending learning materials. This automation can greatly reduce the time and effort required for creating educational materials, enabling educators to dedicate more time to direct teaching and student engagement. However, to fully leverage these benefits, educators must possess the necessary technological, content, and professional expertise to access and evaluate the quality and relevance of the materials provided by ChatGPT.

Malinka et al. [22] studied the impact of ChatGPT on higher education, focusing primarily on computer security-oriented specializations. The study demonstrated the effectiveness and usability of ChatGPT for handling programming assignments, completing exams, and writing term papers. Various levels of tool misuse were evaluated, ranging from using it as a consultant to simply copying its outputs. The benefits of ChatGPT for computer science education were also highlighted.

Rudolph et al. [23] examined the relevance of ChatGPT in the context of higher education, with a particular focus on its impact on assessment, learning, and teaching. Following an overview of ChatGPT's functionality and a summary of its strengths and limitations, the study delved into the broader implications of this technology for the future of higher education. The potential of AI chatbots like ChatGPT to reshape learning, teaching, and assessment practices was explored.

ChatGPT was positioned within the current landscape of AI in education research, with its applications for students, educators, and educational systems discussed. The analysis highlighted both the opportunities and challenges presented by AI integration in education. Practical recommendations were proposed for students, educators, and higher education institutions on how to effectively navigate and leverage these emerging technologies.

Finnie-Ansley et al. [10] likened the introduction of Codex into students' hands to giving a power tool to an amateur—a tool with the potential to either construct or destruct, depending on how it is used. While the presence of AI tools in educational settings presents clear threats to student learning and academic integrity, it also offers fantastic opportunities to refactor existing curricula. By leveraging these tools appropriately, educators can design learning experiences that not only impart essential programming skills but also prepare students for a future where collaboration with AI is an integral part of the software development process.

Traditional teaching methods often suffer from low student participation, lack of personalized instruction, and insufficient motivation [24,25]. Modern approaches in education enhance active learning approaches [26]. Traditional education often relies on theory-based materials, but the integration of ChatGPT can foster personalized learning experiences tailored to individual needs and preferences. Students can leverage ChatGPT to outsource certain knowledge tasks, allowing them to concentrate on 'hands-on' learning and gain practical experience in their chosen fields. Additionally, ChatGPT could also create adaptive learning environments that respond to individual learner progress and performance [14,27,28].

Constructivist theory emphasizes that students construct knowledge through their own experience and participation. Integrating AI tools into a constructivist framework can significantly enhance educational experiences by encouraging students to take charge of their educational journeys through real-time assistance like clarification of concepts, answering questions, and offering tailored guidance based on individual learning needs and styles. With AI tools, educators can create environments that emphasize active learning. Students can engage in inquiry-based activities, utilizing AI as a resource for exploration and problem-solving [29,30,31,32,33]. AI integration may transform educators from content providers to facilitators, emphasizing mentorship and the development of soft skills [4,5,23,27].

Cognitive load theory posits that learning is most effective when cognitive load is managed, helping students focus on essential concepts without being overwhelmed [34,35]. AI tools can reduce extraneous cognitive load by automating routine tasks, allowing students to concentrate on understanding core programming principles. This can be particularly useful for novice programmers who may struggle with complex concepts.

Mandai and co-authors [36], in their opinion paper, discussed the potential impacts of ChatGPT on higher education through the lens of educational theories by John Dewey's Reflective Thought and Action model and the revised Bloom's taxonomy. The analysis was based on a review of existing literature on ChatGPT and educational theories. The key points mentioned positive expectations: ChatGPT could enhance personalized learning, shift education towards more practical, hands-on learning by reducing the need for memorizing information, and lead to assessment reforms. They also mentioned negative expectations, like over-reliance on ChatGPT, failing to acquire necessary skills, diminished student creativity and originality, and they raised concerns about the authenticity and accuracy of sources.

Lo [32], in a literature review, provided an analysis of the implications of ChatGPT in educational contexts. The review assessed ChatGPT's capabilities across various subject domains, its potential applications in education, and the challenges it poses. ChatGPT's performance varied significantly across different subjects, demonstrating outstanding results in critical thinking and economics but showing unsatisfactory performance in mathematics and medical education. Potential applications in education include serving as an assistant for instructors or as a virtual tutor for students. Several challenges were identified, such as accuracy issues, plagiarism concerns, and the need for updated institutional policies to address these issues. The conclusion emphasized that while ChatGPT holds significant promise for enhancing educational practices, careful consideration must be given to its limitations and the ethical implications of its use.

1.2. Aims and Research Questions

As we see, the central question is that of finding the right balance between the different aspects of the use of the tools. To determine the most effective approach, we need a clear understanding of how these tools are actually used in practice, particularly: how students perceive them and how they apply them in real-life programming courses. Especially interesting is observing students' first encounters with these tools, such as during their initial programming course in the first semester of their studies.

In this study, we aim to analyze novice programmers' attitudes toward AI tools. We seek to understand the dynamics of AI tool usage and changes in behavior over time. Additionally, we want to uncover patterns of AI usage and explore students' sentiments about these tools.

Essentially, our goal is to identify the 'Good' and the 'Bad' aspects of AI tool usage in novice programming education. To achieve this, we have formulated the following research questions:

RQ1. How familiar are novice programmers with AI tools during their programming education?

RQ2. Does the usage of AI tools evolve over time in an introductory programming course?

RQ3. Are students satisfied with the results provided by AI tools, and does this satisfaction improve over time?

RQ4. What common tasks do novice programming teams use AI tools for, and how prevalent are these tasks among the teams?

RQ5. What are the common benefits and concerns students have about using AI tools in their studies?

2. Methodology

The primary aims of this study are to understand the dynamics of AI tools usage, uncover patterns in their application, and explore students' sentiments regarding these tools. Data from students' reports and questionnaires on AI tool usage during the Introduction to Programming course are analyzed in this paper. The course spans a period of 12 weeks, with an instructional format consisting of 2 h of lectures and 4 h of lab sessions each week. A pragmatic mixed-methods study was undertaken to explore potential global patterns based on quantitative data, enriched with qualitative data. Students completed surveys and reported on their AI tool usage. The analytical framework of the study is presented in Figure 1.

Education 14 01089 g001

Figure 1. Analytical framework of the study.

2.1. Participants

The participants in this study were first-semester, first-year undergraduate students, who were novice programmers enrolled in the Introduction to Programming course. A total of 73 teams, each comprising two students from the Faculty of Engineering at Ruppin Academic Center in Israel, took part in the study. Students joined a team of their choice and remained with the same team until the end of the semester. Among the participants, 53% were male (78) and 47% were female (68). The median age was 24, with the range between 18 and 29 years. Only teams that submitted all required data items were included in this report. The data from these teams were used for the majority of the analysis in this study.

2.2. Procedure and Data Analysis

All procedures performed in this study were in accordance with APA ethical guidelines and the ethical standards of the institutional research committee. Prior to conducting the research, the requirements of the institutional ethics committee were complied with. At the beginning of the semester, before starting the experimental process, the students were informed about the aims and procedures of the research. The tasks the students would undertake during the experimental process were explained. Each participant was instructed on how to create a unique identifier using a combination of personal but non-identifiable information. This identifier was used in all surveys and bonus questions throughout the semester. Team members merged the two personal, unique identifiers together.

Before the course started, participants completed a pre-semester questionnaire that included questions on students' personal information and basic knowledge of AI. At the end of the course, participants completed a post-semester questionnaire that assessed their perceptions of their AI knowledge, as well as their views on the benefits and concerns related to using AI tools in their studies.

Believing in the importance of incorporating AI tools into classrooms from the very beginning, specific assignment questions were designed where students were tasked with using AI tools to generate or explain portions of code, as well as to explore or learn about particular modules, concepts, or functions. Since these tasks required students to have a basic understanding of AI and the ability to effectively interact with AI tools, several examples were provided to demonstrate how these tools could be used in the context of their assignments. Although no specific AI tool was mandated, students were encouraged to explore different options to find what best suited their needs. As the formulation of the question of the student to the AI tool (usually called the prompt) could be important, in addition to these examples, prompt-writing tips were shared to help students craft more effective queries, enabling them to maximize the potential of the AI tools they chose to work with.

Furthermore, on a weekly basis, students worked on home assignments. A 5-point bonus question related to the usage of AI tools and students' feelings about them was added to some assignments. A bonus question is an additional question or set of questions included in an assignment that offers students the opportunity to earn extra points beyond the regular scoring criteria. To enable continuous data collection throughout the semester and gain insights into how teams interacted with AI tools as they progressed through the course, weekly multipart bonus questions were included in each assignment for the teams to answer. Detailed, ongoing data was gathered, mapping the progression of AI tool integration into their work and providing a nuanced understanding of how their usage evolved.

The course spanned 12 weeks, with weekly assignments categorized into two groups: those requiring the use of AI tools (assignments 3, 7, and 10) and those that did not (assignments 2, 6, 9, and 11). Bonus questions were incorporated into assignments 2, 3, 6, 7, 9, and 11, each having a slightly different structure based on whether AI tools were used. A sample of the bonus question questionnaires can be found in Appendix A. No bonus questions were included in the remaining weeks.

2.3. Data Analysis

The impact of AI tools was measured using several indicators: familiarity with AI tools was assessed through pre- and post-semester questionnaires that captured students' self-reported familiarity levels; the frequency of AI tool usage in assignments was evaluated with a dichotomous (yes/no) statement, "I used AI tools during this assignment", focusing on weeks where usage was not explicitly required to identify trends in voluntary adoption; students' comfort levels with AI tools were tracked during weeks 3, 7, and 10 using a 5-point Likert scale, allowing for the assessment of how comfort evolved as they became more accustomed to the tools; satisfaction with AI tools was measured through survey questions asking students to rate their satisfaction with the results provided by the tools on a Likert scale, offering insights into their perceptions of the quality of outcomes from AI tool usage.

IBM SPSS Statistics 28 software was applied for quantitative data analysis. Several statistical tests were used in this study: the Wilcoxon signed ranks test for two related samples, the Friedman test for several related samples, and Cohran's Q test. Cronbach's alpha coefficients, showing the internal consistency reliability, were computed. The values were acceptable: 0.935 for feeling comfortable with usage of AI tools in the assignment on weeks 3, 7, 10; 0.924 for using AI tools during the assignment on weeks 2, 6, 9; 0.758 for being happy with the results provided by AI tools on weeks 3, 7, 10.

For the qualitative data, participants' responses to the open-ended questions included in the bonus tasks were analyzed using content analysis. Prior to analyzing the data, two sets of categories based on prior research and literature were designed. The first set of categories was designed for responses to the statement, "I used AI tools during this assignment for the following tasks", and the second set was for responses to the direction, "Describe the benefits and concerns about using AI tools in your studies, personally".

We initially employed a deductive coding approach to categorize the data based on these predefined categories. However, several new categories emerged during the analysis, which were subsequently added to the pre-existing codes. Thus, a hybrid coding approach was adopted: the deductive coding provided a structured framework, while the inductive coding allowed for the discovery of additional categories, adding depth and nuance to the findings.

3. Results

Based on the responses received, we analyzed the results to address our research questions. We did not specify the use of particular tools. The majority of students utilized tools designed to understand and generate human-quality text in response to a variety of prompts and questions. Most of these tools are user-friendly, featuring simple interfaces that allow users to input prompts and receive responses easily.

3.1. Familiarity with AI Tools

To answer the first research question about the familiarity of novice programmers with AI tools during their programming education, the level of agreement of the students to the following

statement, “I feel familiar with AI tools usage”, was analyzed at the beginning and at the end of the course. The descriptive frequency and percentage of responses can be found in Table 1. The analysis of survey responses indicated that, initially, only 28% of the teams reported feeling familiar with the usage of AI tools. However, by the end of the course, this familiarity had increased to 100%.

Wilcoxon signed ranks test for two related samples was used to compare the responses at the beginning and at the end of the semester to the statement, “I feel familiar with AI tools usage”, measured on a Likert 1–5 scale. This test revealed significant differences between the responses at the beginning and at the end: $Z = -6.085$, $p < 0.001$.

3.2. Dynamics of AI Tool Integration

To address the second research question—whether the usage of AI tools evolves over time in an introductory programming course—the level of agreement with the statement, “I feel comfortable using AI tools in this assignment”, was analyzed on a Likert scale from 1 to 5 during weeks 3, 7, and 10. During these weeks, the use of AI tools was required to complete the assignments. The descriptive frequencies and percentages of responses are presented in Table 2.

The Friedman test for several related samples was applied, comparing the responses to the item, “I feel comfortable with usage of AI tools in this assignment”, on weeks 3, 7, and 10. The differences in medians were significant: $\chi^2 (2) = 9.8$, $p = 0.007$.

Additionally, to answer the second research question, we analyzed the level of agreement to the dichotomous (yes, no) statement, “I used AI tools during this assignment”, on weeks 2, 6, 9 and 11. Those weeks did not require the use of AI tools. The descriptive frequency and percentage of responses to the statement can be found in Table 3.

Non-parametric Cohran’s Q test was used to compare the response to the item, “I used AI tools during this assignment”, on weeks 2, 6, 9, and 11 (yes, no). Cohran’s $Q = 34.839$, $p < 0.001$, meaning that there were significant differences on the response between weeks 2, 6, 9, and 11, indicating that students used more AI tools.

3.3. Student Satisfaction with AI Tools

To address the third research question, “Are students satisfied with the results provided by AI tools, and does this satisfaction improve over time?”, we analyzed students’ level of agreement with the statement, “I was happy with the results provided by AI tools”, using a Likert scale of 1 to 5 during weeks 3, 7, and 10. During these specific weeks, the use of AI tools was mandatory for completing an assignment. The descriptive frequency and percentage of responses can be found in Table 4.

The Friedman test for several related samples was applied comparing the responses to the item, “I was happy with the results provided by AI tools”, on weeks 3, 7, and 10. The differences in medians were significant: $\chi^2 (2) = 11.594$, $p = 0.003$, meaning that the answers were better. However, since students used the same language model, this result suggests that they improved their prompt-engineering skills over time.

3.4. Common AI Tool Tasks and Prevalence

To address the fourth research question, “What common tasks do novice programming teams use AI tools for, and how prevalent are these tasks among the teams?”, we analyzed the responses to the statement, “I used AI tools during this assignment for the following tasks”. Each team was allowed to provide as many responses as they wished. Given that the quality of

the code and its accompanying description becomes more demanding toward the end of the course, we focused on the teams' responses from weeks 9 and 11. We combined the responses from each team across these two weeks, and if the same category was mentioned more than once, it was counted only once.

Categories that emerged from the analysis of the responses in the qualitative stage of the research and the percentage of the teams that expressed them were as follows: creation of comments mentioned by 67 teams (91.7%), bugs identification and correction—59 teams (80.2%), information seeking—50 teams (68.5%), code comparisons—33 teams (45.21%), creating some part of the code—25 teams (34.2%), debugging assistance—25 teams (34.2%), code simplification—14 teams (19.2%), translation of exercise from Hebrew (the original language) to another language (for example, Arabic or Russian)—12 teams (16.4%), algorithm selection—11 teams (15.1%). Translation of exercise from the original language to another language was a category that was added as a result of the inductive coding approach.

3.5. Benefits and Concerns of AI Tool Usage

To address the fifth research question about the common benefits and concerns students have regarding the use of AI tools in their studies, responses to the open-ended direction, "Describe the benefits and concerns you personally have about using AI tools in your studies", during the last week of the course were analyzed. Common benefit categories emerged from the analysis of the responses, and the percentage of the teams that expressed them were as follows: instant help or quick answers and explanations mentioned by 58 teams (79.4%), assistance with repetitive tasks (like writing comments)—55 teams (75.3%), immediate response/feedback—42 teams (57.5%), help with identification and correction of errors—40 teams (54.8%), access to diverse resources/providing additional examples—33 teams (45.21%), spelling correction—17 teams (23.3%), translation from one language to another—14 teams (19.2%). Once again, translation from one language to another was added during the inductive coding process.

Common concern categories that emerged from the analysis of the responses and the percentage of the teams that expressed them were as follows: inaccurate or misleading information/answers mentioned by 65 teams (89.0%), over-reliance on AI tools/becoming too dependent on AI tools—55 teams (75.3%), identification of advanced functions and structures that were not learned in class—33 teams (45.21%), incorrect explanations which appear believable to novices—27 teams (36.9%), low code quality—25 teams (34.2%), wasting too much time on unsuccessful bug identification and error fixing—12 teams (16.4%), inconsistency/producing different outputs, even when given the same prompt—10 teams (13.6%). This last category was added as a result of the inductive coding approach.

4 Discussion

4.2. The Good

The positive ('Good') aspects can be summarized as follows: AI tools significantly enhanced participants' learning experiences by assisting with information retrieval, bug identification, and writing comments. Participants felt that the use of AI tools brought real-world relevance to the course, fostering essential literacies and skills that are crucial for understanding technology and preparing them for the future. Additionally, some participants noted that receiving feedback from AI tools when they were stuck boosted their self-efficacy and motivation to learn. This finding aligns with Yilmaz and Yilmaz [11], who demonstrated that ChatGPT enhances student

programming self-efficacy and motivation. Similarly, Yin et al. [15] found that chatbot-based learning effectively increases college students' motivation in basic computer science courses. Another advantage ('Good') of incorporating AI tools (beyond just programming courses) is their role in shaping learners' mindsets and skills related to AI, fostering their understanding and application of these technologies. Usage of these tools will help students in their future endeavors to understand the principles of AI, experience AI achievements, and implement AI applications effectively.

4.3. The Bad

On the negative ('Bad') side, challenges emerged, including instances of cheating, where students relied on AI tools to generate entire solutions, particularly under time pressure towards the end of the semester or when facing difficulties with certain learning topics. The efficiency of AI tools also led to the over-automation of some programming tasks, such as generating comments, with 92% of teams fully outsourcing this task to AI. Additionally, there were occasional issues with the quality of produced code, unsuccessful bug identification and error fixing, and the use of incorrect or non-existent functions and packages. AI can generate inaccurate outputs, inaccurate code, or incorrect explanations which appear believable to novices.

The generation of solutions (creating some part of the code) was mentioned only by a third of the teams, possibly because students perceive it as a violation of academic integrity. Despite this, the literature indicates that a high percentage of computer science students engage in some form of plagiarism, with some studies reporting the percentage being nearly 80% [39,40]. Sheard et al. [40] argued that when students are given tasks with readily available solutions in textbooks or lecture notes, they may be tempted to take shortcuts, thus bypassing the intended learning experience. In contrast, Albluwi [41], in his systematic review, noted that research on the relationship between pressure in computing courses and plagiarism is limited and does not adequately reflect the significant impact this factor has on academic dishonesty.

The 'Bad' side of AI tool usage highlighted several significant concerns. One of the primary issues is the risk of students developing an excessive or even blind reliance on these tools, which can lead to a superficial understanding of programming concepts and a diminished ability to solve problems independently. This reliance also opens the door to potential misuse, where the ease of accessibility and inclusion of these tools can tempt students to shortcut their learning process rather than engaging deeply with the material. As Chen et al. [42] mentioned, the value of a tool depends on its use, and there is the potential for Codex to be used in ways that limit learning, or ways that make the work of educators difficult. Actually, the developers of Codex mentioned one such challenge: possible over-reliance on Codex by novice programmers.

Furthermore, the overuse of code generation tools was found to result in a limited grasp of fundamental programming principles and concepts. This not only undermines students' ability to write and understand code on their own but also makes them more vulnerable to technical issues and algorithmic errors. Such errors often arise from the AI's misinterpretation of tasks or from poorly formulated queries, leading to incorrect or suboptimal solutions.

Moreover, instead of fostering a collaborative learning environment where students work alongside AI to enhance their understanding, the AI is often doing most of the work. This dynamic can erode the learning experience, as students become passive recipients rather than

active participants in the problem-solving process. The result is a concerning shift away from critical engagement with programming tasks towards a more superficial reliance on AI-generated solutions.

6. Limitations

This study has some limitations. First, the experimental process was conducted with 73 teams, which introduces the possibility that one team member may have been more dominant in the decision-making process. As a result, the outcomes and responses may reflect the perspectives or actions of that single dominant member, rather than representing the collective input of the entire team. In addition, the sample consists of a specific group of students from one course; therefore, the results might not be representative of all student populations. In further research, the number of participants can be increased, students from additional courses or institutions can join this study, and the wider results can be compared to current ones.

Additionally, a limitation of this study was the lack of direct measures of critical thinking skills. Future research incorporating such measures would provide a more comprehensive understanding of the impact of AI tools on student development. This addition would allow for a more precise evaluation of how AI tools affect students' critical thinking abilities, rather than relying solely on indirect indicators or self-reported data.

Also, much of the data relies on self-reported measures, such as students' comfort levels or perceived improvements. This could introduce biases, as students might overestimate or underestimate their abilities or comfort with AI tools.

--- END PAPER TEXT ---

The Good and Bad of AI Tools in Novice Programming Education

Method: Claim Evidence Extraction

Prompt B:

You are extracting claims and evidence from a research paper about AI-assisted software engineering.

Produce a table with exactly five rows and the following columns:

Claim	Evidence Snippet	Citation (source_id, chunk_id)

Constraints / Guardrails

Each claim must be directly supported by the provided evidence snippet.

Evidence snippets must be copied verbatim or closely paraphrased from the text.

Citations must use the format: (source_id, chunk_id).

Do not invent claims, evidence, or citations.

If the paper does not contain enough information to extract five well-supported claims, include fewer claims and explicitly state “Insufficient evidence for additional claims.”

Do not rely on external knowledge.

Paper: The Good and Bad of AI Tools in Novice Programming Education

--- BEGIN PAPER TEXT ---

Abstract

As AI coding tools become more prevalent in programming, it is essential to understand how they influence programming education. This study, conducted in a first-semester Introduction to Programming course, aimed to determine the positive and negative effects of these tools on students' learning experiences and their ability to develop essential programming skills. Using a mixed-methods approach, we collected data from 73 teams of engineering students over a 12-week period. Students completed surveys and reported on their AI tool usage. We analyzed this data quantitatively to identify trends in tool familiarity, usage, and student satisfaction.

Additionally, qualitative analysis of student reports provided insights into the specific ways AI tools were used and their perceived benefits and drawbacks. The findings revealed a significant increase in AI tool familiarity (from 28% to 100%) and usage among students. Students' satisfaction with AI tools improved over time. The most prevalent tasks for which novice programmers used AI tools included creating comments (91.7%), identifying and correcting bugs (80.2%), and seeking information (68.5%), while other tasks were less common. While these tools offered benefits like assisting in learning and enhancing real-world relevance, they also raised concerns about cheating, over-reliance on AI tools, and a limited understanding of core programming concepts.

Aims and Research Questions

As we see, the central question is that of finding the right balance between the different aspects of the use of the tools. To determine the most effective approach, we need a clear understanding of how these tools are actually used in practice, particularly: how students perceive them and how they apply them in real-life programming courses. Especially interesting is observing students' first encounters with these tools, such as during their initial programming course in the first semester of their studies.

In this study, we aim to analyze novice programmers' attitudes toward AI tools. We seek to understand the dynamics of AI tool usage and changes in behavior over time. Additionally, we want to uncover patterns of AI usage and explore students' sentiments about these tools.

Essentially, our goal is to identify the 'Good' and the 'Bad' aspects of AI tool usage in novice programming education. To achieve this, we have formulated the following research questions:

RQ1. How familiar are novice programmers with AI tools during their programming education?

RQ2. Does the usage of AI tools evolve over time in an introductory programming course?

RQ3. Are students satisfied with the results provided by AI tools, and does this satisfaction improve over time?

RQ4. What common tasks do novice programming teams use AI tools for, and how prevalent are these tasks among the teams?

RQ5. What are the common benefits and concerns students have about using AI tools in their studies?

Methodology

The primary aims of this study are to understand the dynamics of AI tools usage, uncover patterns in their application, and explore students' sentiments regarding these tools. Data from students' reports and questionnaires on AI tool usage during the Introduction to Programming course are analyzed in this paper. The course spans a period of 12 weeks, with an instructional format consisting of 2 h of lectures and 4 h of lab sessions each week. A pragmatic mixed-methods study was undertaken to explore potential global patterns based on quantitative data, enriched with qualitative data. Students completed surveys and reported on their AI tool usage. The analytical framework of the study is presented in Figure 1.

2.1. Participants

The participants in this study were first-semester, first-year undergraduate students, who were novice programmers enrolled in the Introduction to Programming course. A total of 73 teams, each comprising two students from the Faculty of Engineering at Ruppin Academic Center in Israel, took part in the study. Students joined a team of their choice and remained with the same team until the end of the semester. Among the participants, 53% were male (78) and 47% were female (68). The median age was 24, with the range between 18 and 29 years. Only teams that submitted all required data items were included in this report. The data from these teams were used for the majority of the analysis in this study.

2.2. Procedure and Data Analysis

All procedures performed in this study were in accordance with APA ethical guidelines and the ethical standards of the institutional research committee. Prior to conducting the research, the requirements of the institutional ethics committee were complied with. At the beginning of the semester, before starting the experimental process, the students were informed about the aims and procedures of the research. The tasks the students would undertake during the

experimental process were explained. Each participant was instructed on how to create a unique identifier using a combination of personal but non-identifiable information. This identifier was used in all surveys and bonus questions throughout the semester. Team members merged the two personal, unique identifiers together.

Before the course started, participants completed a pre-semester questionnaire that included questions on students' personal information and basic knowledge of AI. At the end of the course, participants completed a post-semester questionnaire that assessed their perceptions of their AI knowledge, as well as their views on the benefits and concerns related to using AI tools in their studies.

Believing in the importance of incorporating AI tools into classrooms from the very beginning, specific assignment questions were designed where students were tasked with using AI tools to generate or explain portions of code, as well as to explore or learn about particular modules, concepts, or functions. Since these tasks required students to have a basic understanding of AI and the ability to effectively interact with AI tools, several examples were provided to demonstrate how these tools could be used in the context of their assignments. Although no specific AI tool was mandated, students were encouraged to explore different options to find what best suited their needs. As the formulation of the question of the student to the AI tool (usually called the prompt) could be important, in addition to these examples, prompt-writing tips were shared to help students craft more effective queries, enabling them to maximize the potential of the AI tools they chose to work with.

Furthermore, on a weekly basis, students worked on home assignments. A 5-point bonus question related to the usage of AI tools and students' feelings about them was added to some assignments. A bonus question is an additional question or set of questions included in an assignment that offers students the opportunity to earn extra points beyond the regular scoring criteria. To enable continuous data collection throughout the semester and gain insights into how teams interacted with AI tools as they progressed through the course, weekly multipart bonus questions were included in each assignment for the teams to answer. Detailed, ongoing data was gathered, mapping the progression of AI tool integration into their work and providing a nuanced understanding of how their usage evolved.

The course spanned 12 weeks, with weekly assignments categorized into two groups: those requiring the use of AI tools (assignments 3, 7, and 10) and those that did not (assignments 2, 6, 9, and 11). Bonus questions were incorporated into assignments 2, 3, 6, 7, 9, and 11, each having a slightly different structure based on whether AI tools were used. A sample of the bonus question questionnaires can be found in Appendix A. No bonus questions were included in the remaining weeks.

first_pageDownload PDFsettingsOrder Article Reprints

Open AccessArticle

The Good and Bad of AI Tools in Novice Programming Education

by Rina Zviel-GirshinORCID

The Center for Research in Technological and Engineering Education, Faculty of Engineering, Ruppin Academic Center, Kfar Monash 4025000, Israel

Educ. Sci. 2024, 14(10), 1089; <https://doi.org/10.3390/educsci14101089>

Submission received: 26 August 2024 / Revised: 29 September 2024 / Accepted: 30 September 2024 / Published: 6 October 2024

(This article belongs to the Special Issue How Artificial Intelligence Can Enhance Education: Current Practices and Challenges)

Download keyboard_arrow_down Browse Figure Review Reports Versions Notes

Abstract

As AI coding tools become more prevalent in programming, it is essential to understand how they influence programming education. This study, conducted in a first-semester Introduction to Programming course, aimed to determine the positive and negative effects of these tools on students' learning experiences and their ability to develop essential programming skills. Using a mixed-methods approach, we collected data from 73 teams of engineering students over a 12-week period. Students completed surveys and reported on their AI tool usage. We analyzed this data quantitatively to identify trends in tool familiarity, usage, and student satisfaction. Additionally, qualitative analysis of student reports provided insights into the specific ways AI tools were used and their perceived benefits and drawbacks. The findings revealed a significant increase in AI tool familiarity (from 28% to 100%) and usage among students. Students' satisfaction with AI tools improved over time. The most prevalent tasks for which novice programmers used AI tools included creating comments (91.7%), identifying and correcting bugs (80.2%), and seeking information (68.5%), while other tasks were less common. While these tools offered benefits like assisting in learning and enhancing real-world relevance, they also raised concerns about cheating, over-reliance on AI tools, and a limited understanding of core programming concepts.

Keywords: artificial intelligence (AI); AI in education; programing education; introduction to programming; AI coding tools

1. Introduction

The integration of Artificial Intelligence (AI) and AI-driven coding tools in programming education has opened new avenues for enhancing both the teaching and learning experience. These tools, while offering significant benefits, can also be a double-edged sword. Our study raises important questions about the AI tools' long-term impact on students' understanding and proficiency. The potential of AI tools to transform education is undeniable. They can automate routine coding tasks, provide instant feedback, improve debugging and troubleshooting, and offer personalized, student-centered learning experiences. However, their use also introduces challenges, particularly regarding the balance between automation and preserving essential programming skills [1,2,3,4,5].

AI in education is rapidly emerging as a key area of focus in educational technology, yet its broader pedagogical implications remain a subject of ongoing debate. The uncertainty about how educators can effectively harness AI to create meaningful learning experiences in higher education is a major concern [3,6,7]. Educators are tasked with the crucial role of preparing students for a future where AI will be ubiquitous, not only in technology-driven industries but across various sectors. To achieve this, educators must equip students with a comprehensive skill set that includes traditional programming principles and knowledge in addition to the ability to leverage cutting-edge AI tools.

Teaching students to use AI tools in programming education aligns with the current demands of the high-tech industry and enhances their readiness for the workplace. By training students to effectively utilize AI tools, educators can improve their efficiency and productivity, allowing them to focus on more complex and creative aspects of software development [8,9]. These tools can

automate tedious and repetitive tasks, thereby freeing up cognitive resources for higher-order problem-solving and innovation [10,11]. However, the integration of AI tools in programming education must be approached with caution. Over-reliance on AI-driven solutions can result in a superficial understanding of programming concepts, where students may become adept at using tools without fully grasping the underlying logic and principles that drive software development [12,13].

A balanced approach is essential, where AI tools are used to complement, rather than replace, traditional coding practices and teaching methods [4,14]. This balance ensures that students develop a deep understanding of programming fundamentals while also learning to use AI to enhance their work. The primary objective of the current study is to investigate how college students use these tools in a first-semester introductory programming course and to explore how educators can achieve this balance, ensuring that AI tools are used to augment learning without diminishing the importance of manual coding and critical thinking [15]. The study also seeks to address the potential risks associated with AI tools, such as the risk of fostering surface-level learning and the possible erosion of essential problem-solving skills.

Ultimately, while AI tools offer significant potential for transforming programming education, their integration must be thoughtfully managed to ensure that they contribute positively to the educational process. By striking the right balance, educators can harness the power of AI to create more dynamic and effective learning environments that prepare students for the challenges of the future.

1.1. Literature Review

Code-generation tools powered by large language models can correctly and reliably solve most of programming problems that are typical in introductory courses. It is therefore unsurprising that Finnie-Ansley et al. [10] showed that AI tools could solve programming assignments and exam problems that are typically given in CS1 (computer science) and CS2 more effectively than most students. The implications of this capability extend beyond mere performance; they challenge the traditional approaches to teaching introductory programming.

Lau and Guo [13], in their semi-structured interviews with introductory programming instructors, revealed a range of perspectives on the integration of AI tools into education. While short-term concerns focused heavily on the potential for cheating, long-term perspectives were more varied. Some educators expressed a desire to resist the use of AI tools in introductory programming courses, concerned that such tools might undermine the development of fundamental skills. On the other hand, others advocated for embracing AI, suggesting that these tools could be integrated into the curriculum to enhance learning and better prepare students for a future where AI is ubiquitous in software development.

The debate about how to manage the influence of AI tools in education is further complicated by the issues of academic integrity and the assessment of student learning. Bommasani et al. [6] emphasized the increasing difficulty educators face in determining the extent of a student's individual contribution. The challenge of preventing ineffective collaborations and detecting plagiarism has become more pronounced as AI tools become more capable. This raises significant questions about the validity of traditional assessment methods and the need for new approaches that can accurately measure a student's understanding and effort in an environment where AI-generated code is easily accessible.

Becker et al. [1] argued in their position paper that AI-generated code presents both opportunities and challenges for students and educators alike. On the one hand, these tools could serve as powerful aids, helping students overcome obstacles in their coding tasks and providing instant feedback that can enhance learning. On the other hand, they pose significant risks, such as the potential for students to become overly reliant on AI-generated solutions, which may hinder the development of critical thinking and problem-solving skills that are central to programming education.

Denny et al. [3] further argued that generative AI brings both challenges and opportunities to computing education, calling for updated pedagogical strategies that focus on new skill sets. The ability of generative AI models to generate solutions to problems typical of introductory programming courses raises concerns about student overreliance and misuse. However, this also opens the door for reimagining the educational landscape, where the focus shifts from rote learning and simple problem-solving to more complex tasks that require higher-order thinking and a deeper understanding of underlying concepts.

In a study conducted by Yilmaz and Yilmaz [11] on undergraduate students, it was found that tools like ChatGPT could enhance students' programming self-efficacy and motivation. The study suggested that ChatGPT could be a valuable tool for teaching computational thinking, providing students with a supportive environment where they can explore programming concepts with instant feedback. This potential for AI tools to positively influence student learning underscores the need for educators to consider how these tools can be integrated into the curriculum in a way that maximizes their benefits while mitigating their risks.

Several studies showed that ChatGPT could provide users with explanations, examples, and guidance to help them understand complex concepts and technologies. It could be used to assist with debugging by analyzing data on the programming language, code structure, error messages, and code documentation, and even assist with code review [16,17,18,19].

Vukojičić's and Krstić's [20] study found that using ChatGPT as a programming assistant resulted in better outcomes than working without external assistance. Students who used ChatGPT demonstrated improved coding proficiency, wrote higher-quality explanations, and gained a deeper understanding of standard solution methods.

Zhai [21] examined ChatGPT from the perspective of educators, asserting that the AI tool holds significant potential for offering specific and personalized guidance to both teachers and student teachers. The study found ChatGPT highly effective in automating essential tasks such as assessment development, grading, learning guidance, and recommending learning materials.

This automation can greatly reduce the time and effort required for creating educational materials, enabling educators to dedicate more time to direct teaching and student engagement. However, to fully leverage these benefits, educators must possess the necessary technological, content, and professional expertise to access and evaluate the quality and relevance of the materials provided by ChatGPT.

Malinka et al. [22] studied the impact of ChatGPT on higher education, focusing primarily on computer security-oriented specializations. The study demonstrated the effectiveness and usability of ChatGPT for handling programming assignments, completing exams, and writing term papers. Various levels of tool misuse were evaluated, ranging from using it as a consultant to simply copying its outputs. The benefits of ChatGPT for computer science education were also highlighted.

Rudolph et al. [23] examined the relevance of ChatGPT in the context of higher education, with a particular focus on its impact on assessment, learning, and teaching. Following an overview of ChatGPT's functionality and a summary of its strengths and limitations, the study delved into the broader implications of this technology for the future of higher education. The potential of AI chatbots like ChatGPT to reshape learning, teaching, and assessment practices was explored. ChatGPT was positioned within the current landscape of AI in education research, with its applications for students, educators, and educational systems discussed. The analysis highlighted both the opportunities and challenges presented by AI integration in education. Practical recommendations were proposed for students, educators, and higher education institutions on how to effectively navigate and leverage these emerging technologies.

Finnie-Ansley et al. [10] likened the introduction of Codex into students' hands to giving a power tool to an amateur—a tool with the potential to either construct or destruct, depending on how it is used. While the presence of AI tools in educational settings presents clear threats to student learning and academic integrity, it also offers fantastic opportunities to refactor existing curricula. By leveraging these tools appropriately, educators can design learning experiences that not only impart essential programming skills but also prepare students for a future where collaboration with AI is an integral part of the software development process.

Traditional teaching methods often suffer from low student participation, lack of personalized instruction, and insufficient motivation [24,25]. Modern approaches in education enhance active learning approaches [26]. Traditional education often relies on theory-based materials, but the integration of ChatGPT can foster personalized learning experiences tailored to individual needs and preferences. Students can leverage ChatGPT to outsource certain knowledge tasks, allowing them to concentrate on 'hands-on' learning and gain practical experience in their chosen fields. Additionally, ChatGPT could also create adaptive learning environments that respond to individual learner progress and performance [14,27,28].

Constructivist theory emphasizes that students construct knowledge through their own experience and participation. Integrating AI tools into a constructivist framework can significantly enhance educational experiences by encouraging students to take charge of their educational journeys through real-time assistance like clarification of concepts, answering questions, and offering tailored guidance based on individual learning needs and styles. With AI tools, educators can create environments that emphasize active learning. Students can engage in inquiry-based activities, utilizing AI as a resource for exploration and problem-solving [29,30,31,32,33]. AI integration may transform educators from content providers to facilitators, emphasizing mentorship and the development of soft skills [4,5,23,27].

Cognitive load theory posits that learning is most effective when cognitive load is managed, helping students focus on essential concepts without being overwhelmed [34,35]. AI tools can reduce extraneous cognitive load by automating routine tasks, allowing students to concentrate on understanding core programming principles. This can be particularly useful for novice programmers who may struggle with complex concepts.

Mandai and co-authors [36], in their opinion paper, discussed the potential impacts of ChatGPT on higher education through the lens of educational theories by John Dewey's Reflective Thought and Action model and the revised Bloom's taxonomy. The analysis was based on a review of existing literature on ChatGPT and educational theories. The key points mentioned positive expectations: ChatGPT could enhance personalized learning, shift education towards

more practical, hands-on learning by reducing the need for memorizing information, and lead to assessment reforms. They also mentioned negative expectations, like over-reliance on ChatGPT, failing to acquire necessary skills, diminished student creativity and originality, and they raised concerns about the authenticity and accuracy of sources.

Lo [32], in a literature review, provided an analysis of the implications of ChatGPT in educational contexts. The review assessed ChatGPT's capabilities across various subject domains, its potential applications in education, and the challenges it poses. ChatGPT's performance varied significantly across different subjects, demonstrating outstanding results in critical thinking and economics but showing unsatisfactory performance in mathematics and medical education. Potential applications in education include serving as an assistant for instructors or as a virtual tutor for students. Several challenges were identified, such as accuracy issues, plagiarism concerns, and the need for updated institutional policies to address these issues. The conclusion emphasized that while ChatGPT holds significant promise for enhancing educational practices, careful consideration must be given to its limitations and the ethical implications of its use.

1.2. Aims and Research Questions

As we see, the central question is that of finding the right balance between the different aspects of the use of the tools. To determine the most effective approach, we need a clear understanding of how these tools are actually used in practice, particularly: how students perceive them and how they apply them in real-life programming courses. Especially interesting is observing students' first encounters with these tools, such as during their initial programming course in the first semester of their studies.

In this study, we aim to analyze novice programmers' attitudes toward AI tools. We seek to understand the dynamics of AI tool usage and changes in behavior over time. Additionally, we want to uncover patterns of AI usage and explore students' sentiments about these tools.

Essentially, our goal is to identify the 'Good' and the 'Bad' aspects of AI tool usage in novice programming education. To achieve this, we have formulated the following research questions:

RQ1. How familiar are novice programmers with AI tools during their programming education?

RQ2. Does the usage of AI tools evolve over time in an introductory programming course?

RQ3. Are students satisfied with the results provided by AI tools, and does this satisfaction improve over time?

RQ4. What common tasks do novice programming teams use AI tools for, and how prevalent are these tasks among the teams?

RQ5. What are the common benefits and concerns students have about using AI tools in their studies?

2. Methodology

The primary aims of this study are to understand the dynamics of AI tools usage, uncover patterns in their application, and explore students' sentiments regarding these tools. Data from students' reports and questionnaires on AI tool usage during the Introduction to Programming course are analyzed in this paper. The course spans a period of 12 weeks, with an instructional format consisting of 2 h of lectures and 4 h of lab sessions each week. A pragmatic mixed-methods study was undertaken to explore potential global patterns based on quantitative data, enriched with qualitative data. Students completed surveys and reported on their AI tool usage. The analytical framework of the study is presented in Figure 1.

Figure 1. Analytical framework of the study.

2.1. Participants

The participants in this study were first-semester, first-year undergraduate students, who were novice programmers enrolled in the Introduction to Programming course. A total of 73 teams, each comprising two students from the Faculty of Engineering at Ruppin Academic Center in Israel, took part in the study. Students joined a team of their choice and remained with the same team until the end of the semester. Among the participants, 53% were male (78) and 47% were female (68). The median age was 24, with the range between 18 and 29 years. Only teams that submitted all required data items were included in this report. The data from these teams were used for the majority of the analysis in this study.

2.2. Procedure and Data Analysis

All procedures performed in this study were in accordance with APA ethical guidelines and the ethical standards of the institutional research committee. Prior to conducting the research, the requirements of the institutional ethics committee were complied with. At the beginning of the semester, before starting the experimental process, the students were informed about the aims and procedures of the research. The tasks the students would undertake during the experimental process were explained. Each participant was instructed on how to create a unique identifier using a combination of personal but non-identifiable information. This identifier was used in all surveys and bonus questions throughout the semester. Team members merged the two personal, unique identifiers together.

Before the course started, participants completed a pre-semester questionnaire that included questions on students' personal information and basic knowledge of AI. At the end of the course, participants completed a post-semester questionnaire that assessed their perceptions of their AI knowledge, as well as their views on the benefits and concerns related to using AI tools in their studies.

Believing in the importance of incorporating AI tools into classrooms from the very beginning, specific assignment questions were designed where students were tasked with using AI tools to generate or explain portions of code, as well as to explore or learn about particular modules, concepts, or functions. Since these tasks required students to have a basic understanding of AI and the ability to effectively interact with AI tools, several examples were provided to demonstrate how these tools could be used in the context of their assignments. Although no specific AI tool was mandated, students were encouraged to explore different options to find what best suited their needs. As the formulation of the question of the student to the AI tool (usually called the prompt) could be important, in addition to these examples, prompt-writing tips were shared to help students craft more effective queries, enabling them to maximize the potential of the AI tools they chose to work with.

Furthermore, on a weekly basis, students worked on home assignments. A 5-point bonus question related to the usage of AI tools and students' feelings about them was added to some assignments. A bonus question is an additional question or set of questions included in an assignment that offers students the opportunity to earn extra points beyond the regular scoring criteria. To enable continuous data collection throughout the semester and gain insights into how teams interacted with AI tools as they progressed through the course, weekly multipart bonus questions were included in each assignment for the teams to answer. Detailed, ongoing

data was gathered, mapping the progression of AI tool integration into their work and providing a nuanced understanding of how their usage evolved.

The course spanned 12 weeks, with weekly assignments categorized into two groups: those requiring the use of AI tools (assignments 3, 7, and 10) and those that did not (assignments 2, 6, 9, and 11). Bonus questions were incorporated into assignments 2, 3, 6, 7, 9, and 11, each having a slightly different structure based on whether AI tools were used. A sample of the bonus question questionnaires can be found in Appendix A. No bonus questions were included in the remaining weeks.

2.3. Data Analysis

The impact of AI tools was measured using several indicators: familiarity with AI tools was assessed through pre- and post-semester questionnaires that captured students' self-reported familiarity levels; the frequency of AI tool usage in assignments was evaluated with a dichotomous (yes/no) statement, "I used AI tools during this assignment", focusing on weeks where usage was not explicitly required to identify trends in voluntary adoption; students' comfort levels with AI tools were tracked during weeks 3, 7, and 10 using a 5-point Likert scale, allowing for the assessment of how comfort evolved as they became more accustomed to the tools; satisfaction with AI tools was measured through survey questions asking students to rate their satisfaction with the results provided by the tools on a Likert scale, offering insights into their perceptions of the quality of outcomes from AI tool usage.

IBM SPSS Statistics 28 software was applied for quantitative data analysis. Several statistical tests were used in this study: the Wilcoxon signed ranks test for two related samples, the Friedman test for several related samples, and Cohran's Q test. Cronbach's alpha coefficients, showing the internal consistency reliability, were computed. The values were acceptable: 0.935 for feeling comfortable with usage of AI tools in the assignment on weeks 3, 7, 10; 0.924 for using AI tools during the assignment on weeks 2, 6, 9; 0.758 for being happy with the results provided by AI tools on weeks 3, 7, 10.

For the qualitative data, participants' responses to the open-ended questions included in the bonus tasks were analyzed using content analysis. Prior to analyzing the data, two sets of categories based on prior research and literature were designed. The first set of categories was designed for responses to the statement, "I used AI tools during this assignment for the following tasks", and the second set was for responses to the direction, "Describe the benefits and concerns about using AI tools in your studies, personally".

We initially employed a deductive coding approach to categorize the data based on these predefined categories. However, several new categories emerged during the analysis, which were subsequently added to the pre-existing codes. Thus, a hybrid coding approach was adopted: the deductive coding provided a structured framework, while the inductive coding allowed for the discovery of additional categories, adding depth and nuance to the findings.

3. Results

Based on the responses received, we analyzed the results to address our research questions. We did not specify the use of particular tools. The majority of students utilized tools designed to understand and generate human-quality text in response to a variety of prompts and questions. Most of these tools are user-friendly, featuring simple interfaces that allow users to input prompts and receive responses easily.

3.1. Familiarity with AI Tools

To answer the first research question about the familiarity of novice programmers with AI tools during their programming education, the level of agreement of the students to the following statement, “I feel familiar with AI tools usage”, was analyzed at the beginning and at the end of the course. The descriptive frequency and percentage of responses can be found in Table 1. The analysis of survey responses indicated that, initially, only 28% of the teams reported feeling familiar with the usage of AI tools. However, by the end of the course, this familiarity had increased to 100%.

Wilcoxon signed ranks test for two related samples was used to compare the responses at the beginning and at the end of the semester to the statement, “I feel familiar with AI tools usage”, measured on a Likert 1–5 scale. This test revealed significant differences between the responses at the beginning and at the end: $Z = -6.085$, $p < 0.001$.

3.2. Dynamics of AI Tool Integration

To address the second research question—whether the usage of AI tools evolves over time in an introductory programming course—the level of agreement with the statement, “I feel comfortable using AI tools in this assignment”, was analyzed on a Likert scale from 1 to 5 during weeks 3, 7, and 10. During these weeks, the use of AI tools was required to complete the assignments. The descriptive frequencies and percentages of responses are presented in Table 2.

The Friedman test for several related samples was applied, comparing the responses to the item, “I feel comfortable with usage of AI tools in this assignment”, on weeks 3, 7, and 10. The differences in medians were significant: $\chi^2 (2) = 9.8$, $p = 0.007$.

Additionally, to answer the second research question, we analyzed the level of agreement to the dichotomous (yes, no) statement, “I used AI tools during this assignment”, on weeks 2, 6, 9 and 11. Those weeks did not require the use of AI tools. The descriptive frequency and percentage of responses to the statement can be found in Table 3.

Non-parametric Cohran’s Q test was used to compare the response to the item, “I used AI tools during this assignment”, on weeks 2, 6, 9, and 11 (yes, no). Cohran’s $Q = 34.839$, $p < 0.001$, meaning that there were significant differences on the response between weeks 2, 6, 9, and 11, indicating that students used more AI tools.

3.3. Student Satisfaction with AI Tools

To address the third research question, “Are students satisfied with the results provided by AI tools, and does this satisfaction improve over time?”, we analyzed students’ level of agreement with the statement, “I was happy with the results provided by AI tools”, using a Likert scale of 1 to 5 during weeks 3, 7, and 10. During these specific weeks, the use of AI tools was mandatory for completing an assignment. The descriptive frequency and percentage of responses can be found in Table 4.

The Friedman test for several related samples was applied comparing the responses to the item, “I was happy with the results provided by AI tools”, on weeks 3, 7, and 10. The differences in medians were significant: $\chi^2 (2) = 11.594$, $p = 0.003$, meaning that the answers were better. However, since students used the same language model, this result suggests that they improved their prompt-engineering skills over time.

3.4. Common AI Tool Tasks and Prevalence

To address the fourth research question, “What common tasks do novice programming teams use AI tools for, and how prevalent are these tasks among the teams?”, we analyzed the

responses to the statement, “I used AI tools during this assignment for the following tasks”. Each team was allowed to provide as many responses as they wished. Given that the quality of the code and its accompanying description becomes more demanding toward the end of the course, we focused on the teams’ responses from weeks 9 and 11. We combined the responses from each team across these two weeks, and if the same category was mentioned more than once, it was counted only once.

Categories that emerged from the analysis of the responses in the qualitative stage of the research and the percentage of the teams that expressed them were as follows: creation of comments mentioned by 67 teams (91.7%), bugs identification and correction—59 teams (80.2%), information seeking—50 teams (68.5%), code comparisons—33 teams (45.21%), creating some part of the code—25 teams (34.2%), debugging assistance—25 teams (34.2%), code simplification—14 teams (19.2%), translation of exercise from Hebrew (the original language) to another language (for example, Arabic or Russian)—12 teams (16.4%), algorithm selection—11 teams (15.1%). Translation of exercise from the original language to another language was a category that was added as a result of the inductive coding approach.

3.5. Benefits and Concerns of AI Tool Usage

To address the fifth research question about the common benefits and concerns students have regarding the use of AI tools in their studies, responses to the open-ended direction, “Describe the benefits and concerns you personally have about using AI tools in your studies”, during the last week of the course were analyzed. Common benefit categories emerged from the analysis of the responses, and the percentage of the teams that expressed them were as follows: instant help or quick answers and explanations mentioned by 58 teams (79.4%), assistance with repetitive tasks (like writing comments)—55 teams (75.3%), immediate response/feedback—42 teams (57.5%), help with identification and correction of errors—40 teams (54.8%), access to diverse resources/providing additional examples—33 teams (45.21%), spelling correction—17 teams (23.3%), translation from one language to another—14 teams (19.2%). Once again, translation from one language to another was added during the inductive coding process.

Common concern categories that emerged from the analysis of the responses and the percentage of the teams that expressed them were as follows: inaccurate or misleading information/answers mentioned by 65 teams (89.0%), over-reliance on AI tools/becoming too dependent on AI tools—55 teams (75.3%), identification of advanced functions and structures that were not learned in class—33 teams (45.21%), incorrect explanations which appear believable to novices—27 teams (36.9%), low code quality—25 teams (34.2%), wasting too much time on unsuccessful bug identification and error fixing—12 teams (16.4%), inconsistency/producing different outputs, even when given the same prompt—10 teams (13.6%). This last category was added as a result of the inductive coding approach.

4 Discussion

4.2. The Good

The positive (“Good”) aspects can be summarized as follows: AI tools significantly enhanced participants’ learning experiences by assisting with information retrieval, bug identification, and writing comments. Participants felt that the use of AI tools brought real-world relevance to the course, fostering essential literacies and skills that are crucial for understanding technology and preparing them for the future. Additionally, some participants noted that receiving feedback from AI tools when they were stuck boosted their self-efficacy and motivation to learn. This finding

aligns with Yilmaz and Yilmaz [11], who demonstrated that ChatGPT enhances student programming self-efficacy and motivation. Similarly, Yin et al. [15] found that chatbot-based learning effectively increases college students' motivation in basic computer science courses. Another advantage ('Good') of incorporating AI tools (beyond just programming courses) is their role in shaping learners' mindsets and skills related to AI, fostering their understanding and application of these technologies. Usage of these tools will help students in their future endeavors to understand the principles of AI, experience AI achievements, and implement AI applications effectively.

4.3. The Bad

On the negative ('Bad') side, challenges emerged, including instances of cheating, where students relied on AI tools to generate entire solutions, particularly under time pressure towards the end of the semester or when facing difficulties with certain learning topics. The efficiency of AI tools also led to the over-automation of some programming tasks, such as generating comments, with 92% of teams fully outsourcing this task to AI. Additionally, there were occasional issues with the quality of produced code, unsuccessful bug identification and error fixing, and the use of incorrect or non-existent functions and packages. AI can generate inaccurate outputs, inaccurate code, or incorrect explanations which appear believable to novices.

The generation of solutions (creating some part of the code) was mentioned only by a third of the teams, possibly because students perceive it as a violation of academic integrity. Despite this, the literature indicates that a high percentage of computer science students engage in some form of plagiarism, with some studies reporting the percentage being nearly 80% [39,40]. Sheard et al. [40] argued that when students are given tasks with readily available solutions in textbooks or lecture notes, they may be tempted to take shortcuts, thus bypassing the intended learning experience. In contrast, Albluwi [41], in his systematic review, noted that research on the relationship between pressure in computing courses and plagiarism is limited and does not adequately reflect the significant impact this factor has on academic dishonesty.

The 'Bad' side of AI tool usage highlighted several significant concerns. One of the primary issues is the risk of students developing an excessive or even blind reliance on these tools, which can lead to a superficial understanding of programming concepts and a diminished ability to solve problems independently. This reliance also opens the door to potential misuse, where the ease of accessibility and inclusion of these tools can tempt students to shortcut their learning process rather than engaging deeply with the material. As Chen et al. [42] mentioned, the value of a tool depends on its use, and there is the potential for Codex to be used in ways that limit learning, or ways that make the work of educators difficult. Actually, the developers of Codex mentioned one such challenge: possible over-reliance on Codex by novice programmers.

Furthermore, the overuse of code generation tools was found to result in a limited grasp of fundamental programming principles and concepts. This not only undermines students' ability to write and understand code on their own but also makes them more vulnerable to technical issues and algorithmic errors. Such errors often arise from the AI's misinterpretation of tasks or from poorly formulated queries, leading to incorrect or suboptimal solutions.

Moreover, instead of fostering a collaborative learning environment where students work alongside AI to enhance their understanding, the AI is often doing most of the work. This

dynamic can erode the learning experience, as students become passive recipients rather than active participants in the problem-solving process. The result is a concerning shift away from critical engagement with programming tasks towards a more superficial reliance on AI-generated solutions.

6. Limitations

This study has some limitations. First, the experimental process was conducted with 73 teams, which introduces the possibility that one team member may have been more dominant in the decision-making process. As a result, the outcomes and responses may reflect the perspectives or actions of that single dominant member, rather than representing the collective input of the entire team. In addition, the sample consists of a specific group of students from one course; therefore, the results might not be representative of all student populations. In further research, the number of participants can be increased, students from additional courses or institutions can join this study, and the wider results can be compared to current ones.

Additionally, a limitation of this study was the lack of direct measures of critical thinking skills. Future research incorporating such measures would provide a more comprehensive understanding of the impact of AI tools on student development. This addition would allow for a more precise evaluation of how AI tools affect students' critical thinking abilities, rather than relying solely on indirect indicators or self-reported data.

Also, much of the data relies on self-reported measures, such as students' comfort levels or perceived improvements. This could introduce biases, as students might overestimate or underestimate their abilities or comfort with AI tools.

--- END PAPER TEXT ---

Paper 4 Prompts

Quick Copy and Paste for:

- [PT Prompt A](#)
- [PT Prompt B](#)
- [CEE Prompt A](#)
- [CEE Prompt B](#)

Measuring GitHub Copilot's Impact on Productivity

Method: Paper Triage

Prompt A:

You are given an academic paper about AI tools in software engineering.

Summarize the paper using the following fields:

Contribution

Method

Data

Findings

Limitations

Paper: Measuring GitHub Copilot's Impact on Productivity

--- BEGIN PAPER TEXT ---

CODE-COMPLETION SYSTEMS OFFERING suggestions to a developer in their integrated development environment (IDE) have become the most frequently used kind of programmer assistance.¹ When generating whole snippets of code, they typically use a large language model (LLM) to predict what the user might type next (the completion) from the context of what they are working on at the moment (the prompt).² This system allows for completions at any position in the code, often spanning multiple lines at once. Potential benefits of generating large sections of code automatically are huge, but evaluating these systems is challenging. Offline evaluation, where the system is shown a partial snippet of code and then asked to complete it, is difficult not least because for longer completions there

are many acceptable alternatives and no straightforward mechanism for labeling them automatically.⁵

An additional step taken by some researchers^{3,21,29} is to use online evaluation and track the frequency of real users accepting suggestions, assuming that the more contributions a system makes to the developer's code, the higher its benefit. The validity of this assumption is not obvious when considering issues such as whether two short completions are more valuable than one long one, or whether reviewing suggestions can be detrimental to programming flow.

Code completion in IDEs using language models was first proposed in Hindle et al.,⁹

and today neural synthesis tools such as GitHub Copilot, CodeWhisperer, and TabNine suggest code snippets within an IDE with the explicitly stated intention to increase a user's productivity. Developer productivity has many aspects, and a recent study has shown that tools like these are helpful in ways that are only partially reflected by measures such as completion times for standardized tasks.^{23,a} Alternatively, we can leverage the developers themselves as expert assessors of their own productivity.

This meshes well with current thinking in software engineering research suggesting measuring productivity on multiple dimensions and using self-reported data.⁶

Thus, we focus on studying perceived productivity.

Here, we investigate whether usage measurements of developer interactions with GitHub Copilot can predict perceived productivity as reported by developers. We analyze 2,631 survey responses from developers using GitHub Copilot and match their responses to measurements collected from the IDE. We consider acceptance counts and more detailed measures of contribution, such as the amount of code contributed by GitHub Copilot and persistence of accepted completions in the code. We find that acceptance rate of shown suggestions is a better predictor of perceived productivity than the alternative measures. We also find that acceptance rate varies significantly over our developer

population as well as over time, and present a deeper dive into some of these variations.

Our results support the principle that acceptance rate can be used for coarse-grained monitoring of the performance of a neural code synthesis system. This ratio of shown suggestions being accepted correlates better than more detailed measures of contribution. However, other approaches remain necessary for fine-grained investigation due to the many human factors involved.

Background

Offline evaluation of code completion can have shortcomings even in tractable circumstances where completions can be labeled for correctness. For example, a study of 15,000 completions by 66 developers in Visual Studio found significant differences between synthetic benchmarks used for model evaluation and real-world usage.⁷

The evaluation of context-aware API completion for Visual Studio IntelliCode considered Recall@5—the proportion of completions for which the correct method call was in the top five suggestions. This metric fell from 90% in offline evaluation to 70% when used online.²¹

Due to the diversity of potential solutions to a multi-line completion task, researchers have used software testing to evaluate the behavior of completions. Competitive programming sites have been used as a source of such data^{8,11} as well as handwritten programming problems.⁵

Yet, it is unclear how well performance on programming competition data generalizes to interactive development in an IDE.

In this work, we define acceptance rate as the fraction of completions shown to the developer that are subsequently accepted for inclusion in the source file. The IntelliCode Compose system uses the term click through rate (CTR) for this and reports a value of 10% in online trials.²⁰ An alternative measure is that of daily completions accepted per user (DCPU) for which a

value of around 20 has been reported.^{3,29} To calculate acceptance rate one must, of course, normalize DCPU by the time spent coding each day. For context, in our study, GitHub Copilot has an acceptance rate of 27% and a mean DCPU in excess of 312 (See Figure 1).^b These differences are presumably due to differences in the kinds of completion offered, or perhaps to user-interface choices. We discuss later how developer objectives, choice of programming language, and even time of day seem to affect our data. Such discrepancies highlight the difficulty in using acceptance rate to understand the value of a system.

There is some evidence that acceptance rate (and indeed correctness) might not tell the whole story. One survey of developers considered the use of AI to support translation between programming languages and found indications that developers tolerated, and in some cases valued, erroneous suggestions from the model.²⁶

Measuring developer productivity through activity counts over time (a typical definition of productivity borrowed from economics) disregards the complexity of software development, as they account for only a subset of developer outputs. A more holistic picture is formed by measuring perceived productivity through self-reported data across various dimensions⁶ and supplementing it with automatically measured data.⁴

We used the SPACE framework⁶ to design a survey that captures self-reported productivity and paired the self-reported data with usage telemetry.

To the best of our knowledge, this is the first study of code suggestion tools establishing a clear link between usage measurements and developer productivity or happiness. A previous study comparing GitHub Copilot against IntelliCode with 25 participants found no significant correlation between task completion times and

survey responses.²²

Data and Methodology

Usage measurements. GitHub Copilot

provides code completions using OpenAI language models. It runs within the IDE and at appropriate points

sends a completion request to a cloudhosted instance of the neural model.

Github Copilot can generate completions at arbitrary points in code rather than, for example, only being triggered when a developer types a period

for invoking a method on an object. A

variety of rules determine when to request a completion, when to abandon requests if the developer has moved

on before the model is ready with a

completion, and how much of the response from the model to surface as a completion.

As stated in our terms of usage,b the

Github Copilot IDE extension records

the events shown in Table 1 for all users. We make usage measurements

for each developer by counting those

events.

Our measures of persistence go further than existing work, which stops at acceptance. The intuition here is that a

completion which is accepted into the source file but then subsequently turns

out to be incorrect can be considered to have wasted developer time both in reviewing it and then having to go back

and delete it. We also record mostly unchanged completions: A large completion requiring a few edits might still be

a positive contribution. It is not clear

how long after acceptance one should

confirm persistence, so we consider a range of options.

The events pertaining to completions form a funnel which we show

quantitatively in Table 1. We include

a summary of all data in Appendix

A.c (All appendices for this article can

be found online at <https://dl.acm.org/>

doi/10.1145/3633453).

We normalize these measures

against each other and write X _

per _ Y to indicate we have normalized metric X by metric Y. For example:

accepted _ per _ hour is calculated as the total number of accepted

events divided by the total number of

(active) hour events.

Table 2 defines the core set of metrics we feel have a natural interpretation in this context. We note there are

alternatives, and we incorporate these in our discussion where relevant.

Productivity survey. To understand users' experience with GitHub Copilot, we emailed a link to an online survey to 17,420 users. These were participants of the unpaid technical preview using GitHub Copilot with their everyday programming tasks.

The only selection criterion was having previously opted in to receive communications. A vast majority of survey users (more than 80%) filled out the survey within the first two days, on or before February 12, 2022. We therefore focus on data from the four-week period leading up to this point ("the study period"). We received a total of 2,047 responses we could match to usage data from the study period, the earliest on Feb. 10, 2022 and the latest on Mar. 6, 2022.

The survey contained multiplechoice questions regarding demographic information (see Figure 2)

and Likert-style questions about different aspects of productivity, which were randomized in their order of appearance to the user. Figure 2 shows the demographic composition of our respondents. We note the significant proportion of professional programmers who responded.

The SPACE framework⁶ defines five dimensions of productivity: Satisfaction and well-being, Performance, Activity, Communication and collaboration, and Efficiency and flow. We use four of these (S,P,C,E), since self reporting on (A) is generally considered inferior to direct measurement. We included 11 statements covering these four dimensions in addition to a single statement: "I am more productive when using GitHub Copilot." For each self-reported productivity measure, we encoded its five ordinal response values to numeric labels (1 = Strongly Disagree, ..., 5 = Strongly Agree). We include the full list of questions and their coding to the SPACE framework in Appendix C. For more information

on the SPACE framework and how the empirical software engineering community has been discussing developer productivity, please see the following section.

Early in our analysis, we found that the usage metrics we describe in the Usage Measurements section corresponded similarly to each of the measured dimensions of productivity, and in turn these dimensions were highly correlated to each other (Figure 3). We therefore added an aggregate productivity score calculated as the mean of all 12 individual measures (excluding skipped questions). This serves as a rough proxy for the much more complex concept of productivity, facilitating recognition of overall trends, which may be less discernible on individual variables due to higher statistical variation. The full dataset of these aggregate productivity scores together with the usage measurements considered in this article is available at <https://bit.ly/47HVjAM>.

Given it has been impossible to produce a unified definition or metric(s) for developer productivity, there have been attempts to synthesize the factors that impact productivity to describe it holistically, include various relevant factors, and treat developer productivity as a composite measure^{17,19,24}. In addition, organizations often use their own multidimensional frameworks to operationalize productivity, which reflects their engineering goals—for example, Google uses the QUANTS framework, with five components of productivity.²⁷ In this article, we use the SPACE framework,⁶ which builds on synthesis of extensive and diverse literature by expert researchers and practitioners in the area of developer productivity. SPACE is an acronym of the five dimensions of productivity:

, S (Satisfaction and well being):
This dimension is meant to reflect developers' fulfillment with the work they do and the tools they use, as well as how healthy and happy they are with the work they do. This dimension reflects some of the easy-to-overlook trade-offs involved when looking exclusively at velocity acceleration—for

example, when we target faster turnaround of code reviews without considering workload impact or burnout for developers.

, P (Performance): This dimension aims to quantify outcomes rather than output. Example metrics that capture performance relate to quality and reliability, as well as further-removed metrics such as customer adoption or satisfaction.

, A (Activity): This is the count of outputs—for example, the number of pull requests closed by a developer. As a result, this dimension is best quantified via system data. Given the variety of developers' activities as part of their work, it is important that the activity dimension accounts for more than coding activity—for instance, writing documentation, creating design specs, and so on.

, C (Communication and collaboration): This dimension aims to capture that modern software development happens in teams and is, therefore, impacted by the discoverability of documentation or the speed of answering questions, or the onboarding time and processing of new team members.

, E (Efficiency and flow): This dimension reflects the ability to complete work or make progress with little interruption or delay. It is important to note that delays and interruptions can be caused either by systems or humans, and it is best to monitor both self-reported and observed measurements—for example, use self-reports of the ability to do uninterrupted work, as well as measure wait time in engineering systems).

What Drives Perceived Productivity?

To examine the relationship between objective measurements of user behavior and self-reported perceptions of productivity, we used our set of core usage measurements (Table 2). We then calculated Pearson's R correlation coefficient and the corresponding p-value of the F-statistic between each pair of usage measurement and perceived productivity metric. We also computed a PLS regression from all usage measurements jointly.

We summarize these results in

Figure 3, showing the correlation coefficients between all measures and

survey questions. The full table of all results is included in Appendix B, available online.

We find acceptance rate (accepted _ per _ shown) most positively predicts users' perception of productivity, although, given the confounding and human factors, there is still notable unexplained variance.

Of all usage measurements, acceptance rate correlates best with aggregate productivity ($p = 0.24$, $P < 0.0001$).

This measurement is also the best performing for at least one survey question in each of the SPACE dimensions.

This correlation is high confidence but leaves considerable unexplained variance. Later, we explore improvements from combining multiple usage measurements together.

Looking at the more detailed metrics around persistence, we see that it is generally better over shorter time periods than over longer periods. This is intuitive in the sense that shorter periods move the measure closer to acceptance rate. We also expect that

at some point after accepting the completion it becomes simply part of the code, so any changes (or not) after that point will not be attributed to GitHub Copilot. All persistence measures were less well correlated than acceptance rate.

To assess the different metrics in a single model, we ran a regression using projection on latent structures (PLS). The choice of PLS, which captures the common variation of these variables as is linearly connected to the aggregate productivity,²⁸ is due to the high collinearity of the single metrics. The first component, to which every metric under consideration contributes positively, explains 43 . 2% of the variance. The second component captures the acceptance rate/change rate dichotomy; it explains a further 13 . 1%. Both draw most strongly from acceptance rate.

This strongly points to acceptance rate being the most immediate indicator of perceived productivity, although it is beneficial to combine with others to get a fuller picture.

Experience

To understand how different types of

developers interact with Copilot, our survey asked respondents to self-report their level of experience in two ways:

- , "Think of the language you have used the most with Copilot. How proficient are you in that language?" with options 'Beginner', 'Intermediate', and 'Advanced'.
- , "Which best describes your programming experience?" with options starting with "Student" and ranging from "0–2 years" to "16+ years" in two year intervals.

We compute correlations with productivity metrics for both experience variables and include these two variables as covariates in a multivariate regression analysis. We find that both are negatively correlated with our aggregate productivity measure (proficiency: $\rho = -0.095$, $P = 0.0001$; years of experience: $\rho = -0.161$, $P < 0.0001$). However, in multivariate regressions predicting productivity from usage metrics while controlling for demographics, proficiency had a non-significant positive effect (coeff = 0.021, $P = 0.213$), while years of experience had a non-significant negative effect (coeff = -0.032 , $P = 0.122$). Looking further at individual measures of productivity, (Table 3) we find that both language proficiency and years of experience negatively predict developers agreeing that Copilot helps them write better code. However, proficiency positively predicts developers agreeing that Copilot helps them stay in the flow, focus on more satisfying work, spend less effort on repetitive tasks, and perform repetitive tasks faster. Years of experience negatively predicts developers feeling less frustrated in coding sessions and performing repetitive tasks faster while using Copilot, but positively predicts developers making progress faster when working in an unfamiliar language. These findings suggest that experienced developers who are already highly skilled are less likely to write better code with Copilot, but

Copilot can assist their productivity in other ways, particularly when engaging with new areas and automating routine work.

Junior developers not only report higher productivity gains; they also tend to accept more suggestions. However, the connection observed in the section "What Drives Perceived Productivity" is not solely due to differing experience levels. In fact, the connection persists in every single experience group, as shown in Figure 5.

Variation over Time

Its connection to perceived productivity motivates a closer look at the acceptance rate and what factors influence

it. Acceptance rate typically increases over the board when the model or underlying prompt-crafting techniques are improved. But even if these conditions are held constant (the study period did not see changes to either), there are more fine-grained temporal patterns emerging.

For coherence of the cultural implications of time of day and weekdays, all data in this section was restricted to users from the U.S. (whether in the survey or not). We used the same time frame as for the investigation in the previous section. In the absence of more fine-grained geolocation, we used the same time zone to interpret timestamps and for day boundaries (PST), recognizing this will introduce some level of noise due to the inhomogeneity of U.S. time zones.

Nevertheless, we observe strong regular patterns in overall acceptance rate (Figure 6). These lead us to distinguish three different time regimes, all of which are statistically significantly distinct at $p < 0.001\%$ (using bootstrap resampling):

- , The weekend: Saturdays and Sundays, where the average acceptance rate is comparatively high at 23.5%.

- , Typical non-working hours during the week: evenings after 4:00 pm PST until mornings 7:00 am PST, where the average acceptance rate is also rather high at 23%.

- , Typical working hours during the week from 7:00 am PST to 4:00 pm PST, where the average acceptance rate is

much lower at 21.2%.

Conclusions

When we set out to connect the productivity benefit of GitHub Copilot to usage measurements from developer activity, we collected measurements about acceptance of completions in line with prior work, but also developed persistence metrics, which arguably capture sustained and direct impact on the resulting code. We were surprised to find acceptance rate (number of acceptances normalized by the number of shown completions) to be better correlated with reported productivity than our measures of persistence.

In hindsight, this makes sense.

Coding is not typing, and GitHub Copilot's central value lies not in being the way users enter most of their code.

Instead, it lies in helping users to make the best progress toward their goals. A suggestion that serves as a useful template to tinker with may be as good or better than a perfectly correct (but obvious) line of code that only saves the user a few keystrokes.

This suggests that a narrow focus on the correctness of suggestions would not tell the whole story for these kinds of tooling. Instead, one could view code suggestions inside an IDE to be more akin to a conversation. While chatbots such as ChatGPT are already used for programming tasks, they are explicitly structured as conversations. Here, we hypothesize that interactions with Copilot, which is not a chatbot, share many characteristics with natural-language conversations.

We see anecdotal evidence of this in comments posted about GitHub Copilot online (see Appendix E for examples), in which users talk about sequences of interactions. A conversation turn in this context consists of the prompt in the completion request and the reply as the completion itself.

The developer's response to the completion arises from the subsequent changes incorporated in the next

prompt to the model. There are clear programming parallels to factors such as specificity and repetition that have been identified to affect human judgements of conversation quality.¹⁸ Researchers have already investigated the benefits of natural-language feedback to guide program synthesis,² so the conversational framing of coding completions is not a radical proposal. But neither is it one we have seen followed yet.

--- END PAPER TEXT ---

Measuring GitHub Copilot's Impact on Productivity

Method: Paper Triage

Prompt B:

You are analyzing an academic paper about AI-assisted tools in software engineering.

Produce a structured summary using exactly the following five fields:

Contribution
Method
Data
Findings
Limitations

Constraints / Guardrails

Base all statements strictly on the content of the provided paper.

Do not infer or speculate beyond what is explicitly supported by the text.

If the paper does not clearly specify information for a field, write: "Not specified in the paper."

Be concise, factual, and neutral in tone.

Do not introduce external knowledge or assumptions.

Output only the five fields listed above.

Paper: Measuring GitHub Copilot's Impact on Productivity

--- BEGIN PAPER TEXT ---

CODE-COMPLETION SYSTEMS OFFERING suggestions to a developer in their integrated development environment (IDE) have become the most frequently used kind of programmer assistance.¹ When generating whole snippets of code, they typically use a large language model (LLM) to predict what the user might type next (the completion) from the context of what they are working on at the moment (the prompt).² This system allows for completions at any position in the code, often spanning multiple lines at once. Potential benefits of generating large sections of code automatically are huge, but evaluating these systems is challenging. Offline evaluation, where the system is shown a partial snippet of code and then asked to complete it, is difficult not least

because for longer completions there are many acceptable alternatives and no straightforward mechanism for labeling them automatically.⁵

An additional step taken by some researchers^{3,21,29} is to use online evaluation and track the frequency of real users accepting suggestions, assuming that the more contributions a system makes to the developer's code, the higher its benefit. The validity of this assumption is not obvious when considering issues such as whether two short completions are more valuable than one long one, or whether reviewing suggestions can be detrimental to programming flow.

Code completion in IDEs using language models was first proposed in Hindle et al.,⁹

and today neural synthesis tools such as GitHub Copilot, CodeWhisperer, and TabNine suggest code snippets within an IDE with the explicitly stated intention to increase a user's productivity. Developer productivity has many aspects, and a recent study has shown that tools like these are helpful in ways that are only partially reflected by measures such as completion times for standardized tasks.^{23,a} Alternatively, we can leverage the developers themselves as expert assessors of their own productivity.

This meshes well with current thinking in software engineering research suggesting measuring productivity on multiple dimensions and using self-reported data.⁶

Thus, we focus on studying perceived productivity.

Here, we investigate whether usage measurements of developer interactions with GitHub Copilot can predict perceived productivity as reported by developers. We analyze 2,631 survey responses from developers using GitHub Copilot and match their responses to measurements collected from the IDE. We consider acceptance counts and more detailed measures of contribution, such as the amount of code contributed by GitHub Copilot and persistence of accepted completions in the code. We find that acceptance rate of shown suggestions is a better predictor of perceived productivity than the alternative measures.

We also find that acceptance rate varies significantly over our developer population as well as over time, and present a deeper dive into some of these variations.

Our results support the principle that acceptance rate can be used for coarse-grained monitoring of the performance of a neural code synthesis system. This ratio of shown suggestions being accepted correlates better than more detailed measures of contribution. However, other approaches remain necessary for fine-grained investigation due to the many human factors involved.

Background

Offline evaluation of code completion can have shortcomings even in tractable circumstances where completions can be labeled for correctness. For example, a study of 15,000 completions by 66 developers in Visual Studio found significant differences between synthetic benchmarks used for model evaluation and real-world usage.⁷

The evaluation of context-aware API completion for Visual Studio IntelliCode considered Recall@5—the proportion of completions for which the correct method call was in the top five suggestions. This metric fell from 90% in offline evaluation to 70% when used online.²¹

Due to the diversity of potential solutions to a multi-line completion task, researchers have used software testing to evaluate the behavior of completions. Competitive programming sites have been used as a source of such data^{8,11} as well as handwritten programming problems.⁵

Yet, it is unclear how well performance on programming competition data generalizes to interactive development in an IDE.

In this work, we define acceptance rate as the fraction of completions shown to the developer that are subsequently accepted for inclusion in the source file. The IntelliCode Compose system uses the term click through rate (CTR) for this and reports a value of 10% in online trials.²⁰ An alternative measure is that of daily completions

accepted per user (DCPU) for which a value of around 20 has been reported.^{3,29} To calculate acceptance rate one must, of course, normalize DCPU by the time spent coding each day. For context, in our study, GitHub Copilot has an acceptance rate of 27% and a mean DCPU in excess of 312 (See Figure 1).^b These differences are presumably due to differences in the kinds of completion offered, or perhaps to user-interface choices. We discuss later how developer objectives, choice of programming language, and even time of day seem to affect our data.

Such discrepancies highlight the difficulty in using acceptance rate to understand the value of a system.

There is some evidence that acceptance rate (and indeed correctness) might not tell the whole story. One survey of developers considered the use of AI to support translation between programming languages and found indications that developers tolerated, and in some cases valued, erroneous suggestions from the model.²⁶

Measuring developer productivity through activity counts over time (a typical definition of productivity borrowed from economics) disregards the complexity of software development, as they account for only a subset of developer outputs. A more holistic picture is formed by measuring perceived productivity through self-reported data across various dimensions⁶ and supplementing it with automatically measured data.⁴

We used the SPACE framework⁶ to design a survey that captures self-reported productivity and paired the self-reported data with usage telemetry.

To the best of our knowledge, this is the first study of code suggestion tools establishing a clear link between usage measurements and developer productivity or happiness. A previous study comparing GitHub Copilot against IntelliCode with 25 participants found no significant correlation

between task completion times and survey responses.²²

Data and Methodology

Usage measurements. GitHub Copilot provides code completions using OpenAI language models. It runs within the IDE and at appropriate points sends a completion request to a cloudhosted instance of the neural model. GitHub Copilot can generate completions at arbitrary points in code rather than, for example, only being triggered when a developer types a period for invoking a method on an object. A variety of rules determine when to request a completion, when to abandon requests if the developer has moved on before the model is ready with a completion, and how much of the response from the model to surface as a completion.

As stated in our terms of usage,^b the GitHub Copilot IDE extension records the events shown in Table 1 for all users. We make usage measurements for each developer by counting those events.

Our measures of persistence go further than existing work, which stops at acceptance. The intuition here is that a completion which is accepted into the source file but then subsequently turns out to be incorrect can be considered to have wasted developer time both in reviewing it and then having to go back and delete it. We also record mostly unchanged completions: A large completion requiring a few edits might still be a positive contribution. It is not clear how long after acceptance one should confirm persistence, so we consider a range of options.

The events pertaining to completions form a funnel which we show quantitatively in Table 1. We include a summary of all data in Appendix A.c (All appendices for this article can be found online at <https://dl.acm.org/doi/10.1145/3633453>).

We normalize these measures against each other and write X_per_Y to indicate we have normalized metric X by metric Y. For example: accepted_per_hour is calculated as the total number of accepted events divided by the total number of

(active) hour events.

Table 2 defines the core set of metrics we feel have a natural interpretation in this context. We note there are

alternatives, and we incorporate these in our discussion where relevant.

Productivity survey. To understand

users' experience with GitHub Copilot, we emailed a link to an online

survey to 17, 420 users. These were

participants of the unpaid technical

preview using GitHub Copilot with

their everyday programming tasks.

The only selection criterion was having previously opted in to receive communications. A vast majority of survey

users (more than 80%) filled out the

survey within the first two days, on or

before February 12, 2022. We therefore focus on data from the four-week

period leading up to this point ("the

study period"). We received a total of

2,047 responses we could match to

usage data from the study period, the

earliest on Feb. 10, 2022 and the latest

on Mar. 6, 2022.

The survey contained multiplechoice questions regarding demographic information (see Figure 2)

and Likert-style questions about different aspects of productivity, which were randomized in their order of appearance to the user. Figure 2 shows the demographic composition of our respondents. We note the significant proportion of professional programmers who responded.

The SPACE framework⁶

defines five

dimensions of productivity: Satisfaction and well-being, Performance, Activity, Communication and collaboration, and Efficiency and flow. We use

four of these (S,P,C,E), since self reporting on (A) is generally considered inferior to direct measurement. We

included 11 statements covering these

four dimensions in addition to a single statement: "I am more productive

when using GitHub Copilot." For each

self-reported productivity measure,

we encoded its five ordinal response

values to numeric labels (1 = Strongly

Disagree, ..., 5 = Strongly Agree). We

include the full list of questions and

their coding to the SPACE framework

in Appendix C. For more information on the SPACE framework and how the empirical software engineering community has been discussing developer productivity, please see the following section.

Early in our analysis, we found that the usage metrics we describe in the Usage Measurements section corresponded similarly to each of the measured dimensions of productivity, and in turn these dimensions were highly correlated to each other (Figure 3). We therefore added an aggregate productivity score calculated as the mean of all 12 individual measures (excluding skipped questions). This serves as a rough proxy for the much more complex concept of productivity, facilitating recognition of overall trends, which may be less discernible on individual variables due to higher statistical variation. The full dataset of these aggregate productivity scores together with the usage measurements considered in this article is available at <https://bit.ly/47HVjAM>.

Given it has been impossible to produce a unified definition or metric(s) for developer productivity, there have been attempts to synthesize the factors that impact productivity to describe it holistically, include various relevant factors, and treat developer productivity as a composite measure^{17,19,24}. In addition, organizations often use their own multidimensional frameworks to operationalize productivity, which reflects their engineering goals—for example, Google uses the QUANTS framework, with five components of productivity.²⁷ In this article, we use the SPACE framework,⁶ which

builds on synthesis of extensive and diverse literature by expert researchers and practitioners in the area of developer productivity. SPACE is an acronym of the five dimensions of productivity:

, S (Satisfaction and well being):

This dimension is meant to reflect developers' fulfillment with the work they do and the tools they use, as well as how healthy and happy they are with the work they do. This dimension reflects some of the easy-to-overlook trade-offs involved when looking exclusively at velocity acceleration—for

example, when we target faster turnaround of code reviews without considering workload impact or burnout for developers.

, P (Performance): This dimension aims to quantify outcomes rather than output. Example metrics that capture performance relate to quality and reliability, as well as further-removed metrics such as customer adoption or satisfaction.

, A (Activity): This is the count of outputs—for example, the number of pull requests closed by a developer. As a result, this dimension is best quantified via system data. Given the variety of developers' activities as part of their work, it is important that the activity dimension accounts for more than coding activity—for instance, writing documentation, creating design specs, and so on.

, C (Communication and collaboration): This dimension aims to capture that modern software development happens in teams and is, therefore, impacted by the discoverability of documentation or the speed of answering questions, or the onboarding time and processing of new team members.

, E (Efficiency and flow): This dimension reflects the ability to complete work or make progress with little interruption or delay. It is important to note that delays and interruptions can be caused either by systems or humans, and it is best to monitor both self-reported and observed measurements—for example, use self-reports of the ability to do uninterrupted work, as well as measure wait time in engineering systems).

What Drives Perceived Productivity?

To examine the relationship between objective measurements of user behavior and self-reported perceptions of productivity, we used our set of core usage measurements (Table 2). We then calculated Pearson's R correlation coefficient and the corresponding p-value of the F-statistic between each pair of usage measurement and perceived productivity metric. We also computed a PLS regression from all usage measurements jointly.

We summarize these results in

Figure 3, showing the correlation coefficients between all measures and

survey questions. The full table of all results is included in Appendix B, available online.

We find acceptance rate (accepted _ per _ shown) most positively predicts users' perception of productivity, although, given the confounding and human factors, there is still notable unexplained variance.

Of all usage measurements, acceptance rate correlates best with aggregate productivity ($p = 0.24$, $P < 0.0001$).

This measurement is also the best performing for at least one survey question in each of the SPACE dimensions.

This correlation is high confidence but leaves considerable unexplained variance. Later, we explore improvements from combining multiple usage measurements together.

Looking at the more detailed metrics around persistence, we see that it is generally better over shorter time periods than over longer periods. This is intuitive in the sense that shorter periods move the measure closer to acceptance rate. We also expect that

at some point after accepting the completion it becomes simply part of the code, so any changes (or not) after that point will not be attributed to GitHub Copilot. All persistence measures were less well correlated than acceptance rate.

To assess the different metrics in a single model, we ran a regression using projection on latent structures (PLS). The choice of PLS, which captures the common variation of these variables as is linearly connected to the aggregate productivity,²⁸ is due to the high collinearity of the single metrics. The first component, to which every metric under consideration contributes positively, explains 43 . 2% of the variance. The second component captures the acceptance rate/change rate dichotomy; it explains a further 13 . 1%. Both draw most strongly from acceptance rate.

This strongly points to acceptance rate being the most immediate indicator of perceived productivity, although it is beneficial to combine with others to get a fuller picture.

Experience

To understand how different types of

developers interact with Copilot, our survey asked respondents to self-report their level of experience in two ways:

- , "Think of the language you have used the most with Copilot. How proficient are you in that language?" with options 'Beginner', 'Intermediate', and 'Advanced'.
- , "Which best describes your programming experience?" with options starting with "Student" and ranging from "0–2 years" to "16+ years" in two-year intervals.

We compute correlations with productivity metrics for both experience variables and include these two variables as covariates in a multivariate regression analysis. We find that both are negatively correlated with our aggregate productivity measure (proficiency: $\rho = -0.095$, $P = 0.0001$; years of experience: $\rho = -0.161$, $P < 0.0001$). However, in multivariate regressions predicting productivity from usage metrics while controlling for demographics, proficiency had a non-significant positive effect (coeff = 0.021, $P = 0.213$), while years of experience had a non-significant negative effect (coeff = -0.032 , $P = 0.122$). Looking further at individual measures of productivity, (Table 3) we find that both language proficiency and years of experience negatively predict developers agreeing that Copilot helps them write better code. However, proficiency positively predicts developers agreeing that Copilot helps them stay in the flow, focus on more satisfying work, spend less effort on repetitive tasks, and perform repetitive tasks faster. Years of experience negatively predicts developers feeling less frustrated in coding sessions and performing repetitive tasks faster while using Copilot, but positively predicts developers making progress faster when working in an unfamiliar language. These findings suggest that experienced developers who are already highly skilled are less likely to write better code with Copilot, but

Copilot can assist their productivity in other ways, particularly when engaging with new areas and automating routine work.

Junior developers not only report higher productivity gains; they also tend to accept more suggestions. However, the connection observed in the section "What Drives Perceived Productivity" is not solely due to differing experience levels. In fact, the connection persists in every single experience group, as shown in Figure 5.

Variation over Time

Its connection to perceived productivity motivates a closer look at the acceptance rate and what factors influence

it. Acceptance rate typically increases over the board when the model or underlying prompt-crafting techniques are improved. But even if these conditions are held constant (the study period did not see changes to either), there are more fine-grained temporal patterns emerging.

For coherence of the cultural implications of time of day and weekdays, all data in this section was restricted to users from the U.S. (whether in the survey or not). We used the same time frame as for the investigation in the previous section. In the absence of more fine-grained geolocation, we used the same time zone to interpret timestamps and for day boundaries (PST), recognizing this will introduce some level of noise due to the inhomogeneity of U.S. time zones.

Nevertheless, we observe strong regular patterns in overall acceptance rate (Figure 6). These lead us to distinguish three different time regimes, all of which are statistically significantly distinct at $p < 0.001\%$ (using bootstrap resampling):

- , The weekend: Saturdays and Sundays, where the average acceptance rate is comparatively high at 23.5%.

- , Typical non-working hours during the week: evenings after 4:00 pm PST until mornings 7:00 am PST, where the average acceptance rate is also rather high at 23%.

- , Typical working hours during the week from 7:00 am PST to 4:00 pm PST, where the average acceptance rate is

much lower at 21.2%.

Conclusions

When we set out to connect the productivity benefit of GitHub Copilot to usage measurements from developer activity, we collected measurements about acceptance of completions in line with prior work, but also developed persistence metrics, which arguably capture sustained and direct impact on the resulting code. We were surprised to find acceptance rate (number of acceptances normalized by the number of shown completions) to be better correlated with reported productivity than our measures of persistence.

In hindsight, this makes sense.

Coding is not typing, and GitHub Copilot's central value lies not in being the way users enter most of their code.

Instead, it lies in helping users to make the best progress toward their goals. A suggestion that serves as a useful template to tinker with may be as good or better than a perfectly correct (but obvious) line of code that only saves the user a few keystrokes.

This suggests that a narrow focus on the correctness of suggestions would not tell the whole story for these kinds of tooling. Instead, one could view code suggestions inside an IDE to be more akin to a conversation. While chatbots such as ChatGPT are already used for programming tasks, they are explicitly structured as conversations. Here, we hypothesize that interactions with Copilot, which is not a chatbot, share many characteristics with natural-language conversations.

We see anecdotal evidence of this in comments posted about GitHub Copilot online (see Appendix E for examples), in which users talk about sequences of interactions. A conversation turn in this context consists of the prompt in the completion request and the reply as the completion itself.

The developer's response to the completion arises from the subsequent changes incorporated in the next

prompt to the model. There are clear programming parallels to factors such as specificity and repetition that have been identified to affect human judgements of conversation quality.¹⁸ Researchers have already investigated the benefits of natural-language feedback to guide program synthesis,² so the conversational framing of coding completions is not a radical proposal. But neither is it one we have seen followed yet.

--- END PAPER TEXT ---

Measuring GitHub Copilot's Impact on Productivity

Method: Claim Evidence Extraction

Prompt A:

From the provided paper, extract five key claims about the impact of AI tools on software engineering.

For each claim, include:

The claim

A supporting quote or snippet from the text

A citation

Present your answer in a table.

Paper: Measuring GitHub Copilot's Impact on Productivity

--- BEGIN PAPER TEXT ---

CODE-COMPLETION SYSTEMS OFFERING suggestions to a developer in their integrated development environment (IDE) have become the most frequently used kind of programmer assistance.¹ When generating whole snippets of code, they typically use a large language model (LLM) to predict what the user might type next (the completion) from the context of what they are working on at the moment (the prompt).² This system allows for completions at any position in the code, often spanning multiple lines at once. Potential benefits of generating large sections of code automatically are huge, but evaluating these systems is challenging. Offline evaluation, where the system is shown a partial snippet of code and then asked to complete it, is difficult not least because for longer completions there are many acceptable alternatives and no straightforward mechanism for labeling them automatically.⁵ An additional step taken by some researchers^{3,21,29} is to use online evaluation and track the frequency of real users accepting suggestions, assuming that the more contributions a system makes to the developer's code, the

higher its benefit. The validity of this assumption is not obvious when considering issues such as whether two short completions are more valuable than one long one, or whether reviewing suggestions can be detrimental to programming flow.

Code completion in IDEs using language models was first proposed in Hindle et al.,⁹ and today neural synthesis tools such as GitHub Copilot, CodeWhisperer, and TabNine suggest code snippets within an IDE with the explicitly stated intention to increase a user's productivity. Developer productivity has many aspects, and a recent study has shown that tools like these are helpful in ways that are only partially reflected by measures such as completion times for standardized tasks.^{23,a} Alternatively, we can leverage the developers themselves as expert assessors of their own productivity. This meshes well with current thinking in software engineering research suggesting measuring productivity on multiple dimensions and using self-reported data.⁶

Thus, we focus on studying perceived productivity. Here, we investigate whether usage measurements of developer interactions with GitHub Copilot can predict perceived productivity as reported by developers. We analyze 2,631 survey responses from developers using GitHub Copilot and match their responses to measurements collected from the IDE. We consider acceptance counts and more detailed measures of contribution, such as the amount of code contributed by GitHub Copilot and persistence of accepted completions in the code. We find that acceptance rate of shown suggestions is a better predictor of perceived productivity than the alternative measures. We also find that acceptance rate varies significantly over our developer population as well as over time, and present a deeper dive into some of these variations.

Our results support the principle that acceptance rate can be used for coarse-grained monitoring of the performance of a neural code synthesis system. This ratio of shown suggestions being accepted correlates better

than more detailed measures of contribution. However, other approaches remain necessary for fine-grained investigation due to the many human factors involved.

Background

Offline evaluation of code completion can have shortcomings even in tractable circumstances where completions can be labeled for correctness. For example, a study of 15,000 completions by 66 developers in Visual Studio found significant differences between synthetic benchmarks used for model evaluation and real-world usage.⁷

The evaluation

of context-aware API completion for Visual Studio IntelliCode considered Recall@5—the proportion of completions for which the correct method call was in the top five suggestions. This metric fell from 90% in offline evaluation to 70% when used online.²¹

Due to the diversity of potential solutions to a multi-line completion task, researchers have used software testing to evaluate the behavior of completions. Competitive programming sites have been used as a source of such data^{8,11} as well as handwritten programming problems.⁵

Yet, it is unclear how well performance on programming competition data generalizes to interactive development in an IDE.

In this work, we define acceptance rate as the fraction of completions shown to the developer that are subsequently accepted for inclusion in the source file. The IntelliCode Compose system uses the term click through rate (CTR) for this and reports a value of 10% in online trials.²⁰ An alternative measure is that of daily completions accepted per user (DCPU) for which a value of around 20 has been reported.^{3,29} To calculate acceptance rate one must, of course, normalize DCPU by the time spent coding each day. For context, in our study, GitHub Copilot has an acceptance rate of 27% and a mean DCPU in excess of 312 (See Figure 1).^b These differences are presumably due to differences in the kinds

of completion offered, or perhaps to user-interface choices. We discuss later how developer objectives, choice of programming language, and even time of day seem to affect our data.

Such discrepancies highlight the difficulty in using acceptance rate to understand the value of a system.

There is some evidence that acceptance rate (and indeed correctness) might not tell the whole story. One survey of developers considered the use of AI to support translation between programming languages and found indications that developers tolerated, and in some cases valued, erroneous suggestions from the model.²⁶

Measuring developer productivity through activity counts over time (a typical definition of productivity borrowed from economics) disregards the complexity of software development, as they account for only a subset of developer outputs. A more holistic picture is formed by measuring perceived productivity through self-reported data across various dimensions⁶

and supplementing it with automatically measured data.⁴

We used the SPACE framework⁶

to design a survey that captures self-reported productivity and paired the self-reported data with usage telemetry.

To the best of our knowledge, this is the first study of code suggestion tools establishing a clear link between usage measurements and developer productivity or happiness. A previous study comparing GitHub Copilot against IntelliCode with 25 participants found no significant correlation between task completion times and survey responses.²²

Data and Methodology

Usage measurements. GitHub Copilot provides code completions using OpenAI language models. It runs within the IDE and at appropriate points sends a completion request to a cloudhosted instance of the neural model.

GitHub Copilot can generate completions at arbitrary points in code rather than, for example, only being triggered when a developer types a period for invoking a method on an object. A variety of rules determine when to request a completion, when to abandon requests if the developer has moved on before the model is ready with a completion, and how much of the response from the model to surface as a completion.

As stated in our terms of usage,^b the GitHub Copilot IDE extension records the events shown in Table 1 for all users. We make usage measurements for each developer by counting those events.

Our measures of persistence go further than existing work, which stops at acceptance. The intuition here is that a completion which is accepted into the source file but then subsequently turns out to be incorrect can be considered to have wasted developer time both in reviewing it and then having to go back and delete it. We also record mostly unchanged completions: A large completion requiring a few edits might still be a positive contribution. It is not clear how long after acceptance one should confirm persistence, so we consider a range of options.

The events pertaining to completions form a funnel which we show quantitatively in Table 1. We include a summary of all data in Appendix A.c (All appendices for this article can be found online at <https://dl.acm.org/doi/10.1145/3633453>).

We normalize these measures against each other and write X_per_Y to indicate we have normalized metric X by metric Y. For example: accepted_per_hour is calculated as the total number of accepted events divided by the total number of (active) hour events.

Table 2 defines the core set of metrics we feel have a natural interpretation in this context. We note there are alternatives, and we incorporate these in our discussion where relevant. Productivity survey. To understand users' experience with GitHub Copilot, we emailed a link to an online

survey to 17,420 users. These were participants of the unpaid technical preview using GitHub Copilot with their everyday programming tasks.

The only selection criterion was having previously opted in to receive communications. A vast majority of survey users (more than 80%) filled out the survey within the first two days, on or before February 12, 2022. We therefore focus on data from the four-week period leading up to this point ("the study period"). We received a total of 2,047 responses we could match to usage data from the study period, the earliest on Feb. 10, 2022 and the latest on Mar. 6, 2022.

The survey contained multiplechoice questions regarding demographic information (see Figure 2)

and Likert-style questions about different aspects of productivity, which were randomized in their order of appearance to the user. Figure 2 shows the demographic composition of our respondents. We note the significant proportion of professional programmers who responded.

The SPACE framework⁶ defines five dimensions of productivity: Satisfaction and well-being, Performance, Activity, Communication and collaboration, and Efficiency and flow. We use four of these (S,P,C,E), since self reporting on (A) is generally considered inferior to direct measurement. We included 11 statements covering these four dimensions in addition to a single statement: "I am more productive when using GitHub Copilot." For each self-reported productivity measure, we encoded its five ordinal response values to numeric labels (1 = Strongly Disagree, ..., 5 = Strongly Agree). We include the full list of questions and their coding to the SPACE framework in Appendix C. For more information on the SPACE framework and how the empirical software engineering community has been discussing developer productivity, please see the following section.

Early in our analysis, we found that the usage metrics we describe in the

Usage Measurements section corresponded similarly to each of the measured dimensions of productivity, and in turn these dimensions were highly correlated to each other (Figure 3). We therefore added an aggregate productivity score calculated as the mean of all 12 individual measures (excluding skipped questions). This serves as a rough proxy for the much more complex concept of productivity, facilitating recognition of overall trends, which may be less discernible on individual variables due to higher statistical variation. The full dataset of these aggregate productivity scores together with the usage measurements considered in this article is available at <https://bit.ly/47HVjAM>.

Given it has been impossible to produce a unified definition or metric(s) for developer productivity, there have been attempts to synthesize the factors that impact productivity to describe it holistically, include various relevant factors, and treat developer productivity as a composite measure^{17,19,24}. In addition, organizations often use their own multidimensional frameworks to operationalize productivity, which reflects their engineering goals—for example, Google uses the QUANTS framework, with five components of productivity.²⁷ In this article, we use the SPACE framework,⁶ which

builds on synthesis of extensive and diverse literature by expert researchers and practitioners in the area of developer productivity. SPACE is an acronym of the five dimensions of productivity:

, S (Satisfaction and well being):

This dimension is meant to reflect developers' fulfillment with the work they do and the tools they use, as well as how healthy and happy they are with the work they do. This dimension reflects some of the easy-to-overlook trade-offs involved when looking exclusively at velocity acceleration—for example, when we target faster turnaround of code reviews without considering workload impact or burnout for developers.

, P (Performance): This dimension aims to quantify outcomes rather than output. Example metrics that capture performance relate to quality and reliability, as well as further-removed

metrics such as customer adoption or satisfaction.

, A (Activity): This is the count of outputs—for example, the number of pull requests closed by a developer. As a result, this dimension is best quantified via system data. Given the variety of developers' activities as part of their work, it is important that the activity dimension accounts for more than coding activity—for instance, writing documentation, creating design specs, and so on.

, C (Communication and collaboration): This dimension aims to capture that modern software development happens in teams and is, therefore, impacted by the discoverability of documentation or the speed of answering questions, or the onboarding time and processing of new team members.

, E (Efficiency and flow): This dimension reflects the ability to complete work or make progress with little interruption or delay. It is important to note that delays and interruptions can be caused either by systems or humans, and it is best to monitor both self-reported and observed measurements—for example, use self-reports of the ability to do uninterrupted work, as well as measure wait time in engineering systems).

What Drives Perceived Productivity?

To examine the relationship between objective measurements of user behavior and self-reported perceptions of productivity, we used our set of core usage measurements (Table 2). We then calculated Pearson's R correlation coefficient and the corresponding p-value of the F-statistic between each pair of usage measurement and perceived productivity metric. We also computed a PLS regression from all usage measurements jointly.

We summarize these results in

Figure 3, showing the correlation coefficients between all measures and survey questions. The full table of all results is included in Appendix B, available online.

We find acceptance rate (accepted _ per _ shown) most positively predicts users' perception of productivity, although, given the confounding and human factors, there is still notable unexplained variance.

Of all usage measurements, acceptance rate correlates best with aggregate productivity ($p = 0.24$, $P < 0.0001$).

This measurement is also the best performing for at least one survey question in each of the SPACE dimensions.

This correlation is high confidence but leaves considerable unexplained variance. Later, we explore improvements from combining multiple usage measurements together.

Looking at the more detailed metrics around persistence, we see that it is generally better over shorter time periods than over longer periods. This is intuitive in the sense that shorter periods move the measure closer to acceptance rate. We also expect that at some point after accepting the completion it becomes simply part of the code, so any changes (or not) after that point will not be attributed to GitHub Copilot. All persistence measures were less well correlated than acceptance rate.

To assess the different metrics in a single model, we ran a regression using projection on latent structures (PLS). The choice of PLS, which captures the common variation of these variables as is linearly connected to the aggregate productivity,²⁸ is due to the high collinearity of the single metrics. The first component, to which every metric under consideration contributes positively, explains 43 . 2% of the variance. The second component captures the acceptance rate/change rate dichotomy; it explains a further 13 . 1%. Both draw most strongly from acceptance rate.

This strongly points to acceptance rate being the most immediate indicator of perceived productivity, although it is beneficial to combine with others to get a fuller picture.

Experience

To understand how different types of developers interact with Copilot, our survey asked respondents to self-report their level of experience in two ways:

· "Think of the language you have used the most with Copilot. How proficient are you in that language?" with options 'Beginner', 'Intermediate', and 'Advanced'.

, "Which best describes your programming experience?" with options starting with "Student" and ranging from "0–2 years" to "16+ years" in twoyear intervals.

We compute correlations with productivity metrics for both experience variables and include these two variables as covariates in a multivariate regression analysis. We find that both are

negatively correlated with our aggregate productivity measure (proficiency: $p = -0.095$, $P = 0.0001$; years of experience: $p = -0.161$, $P < 0.0001$).

However, in multivariate regressions predicting productivity from usage metrics while controlling for demographics, proficiency had a non-significant positive effect (coeff = 0.021,

$P = 0.213$), while years of experience had a non-significant negative effect (coeff = -0.032, $P = 0.122$).

Looking further at individual measures of productivity, (Table 3) we find that both language proficiency and years of experience negatively predict developers agreeing that Copilot helps them write better code. However, proficiency positively predicts developers agreeing that Copilot helps them stay in the flow, focus on more satisfying work, spend less effort on repetitive tasks, and perform repetitive tasks faster. Years of experience negatively predicts developers feeling less frustrated in coding sessions and performing repetitive tasks faster while

using Copilot, but positively predicts developers making progress faster when working in an unfamiliar language. These findings suggest that

experienced developers who are already highly skilled are less likely to write better code with Copilot, but Copilot can assist their productivity in other ways, particularly when engaging with new areas and automating routine work.

Junior developers not only report higher productivity gains; they also tend to accept more suggestions. However, the connection observed in the section "What Drives Perceived Productivity" is not solely due to differing

experience levels. In fact, the connection persists in every single experience group, as shown in Figure 5.

Variation over Time

Its connection to perceived productivity motivates a closer look at the acceptance rate and what factors influence

it. Acceptance rate typically increases

over the board when the model or underlying prompt-crafting techniques are improved. But even if these conditions are held constant (the study period did not see changes to either),

there are more fine-grained temporal patterns emerging.

For coherence of the cultural implications of time of day and weekdays,

all data in this section was restricted

to users from the U.S. (whether in

the survey or not). We used the same

time frame as for the investigation in

the previous section. In the absence

of more fine-grained geolocation, we

used the same time zone to interpret

timestamps and for day boundaries

(PST), recognizing this will introduce

some level of noise due to the inhomogeneity of U.S. time zones.

Nevertheless, we observe strong

regular patterns in overall acceptance

rate (Figure 6). These lead us to distinguish three different time regimes, all

of which are statistically significantly

distinct at $p < 0.001\%$ (using bootstrap resampling):

, The weekend: Saturdays and Sundays, where the average acceptance rate is comparatively high at 23.5%.

, Typical non-working hours during the week: evenings after 4:00 pm PST until mornings 7:00 am PST, where the average acceptance rate is also rather high at 23%.

, Typical working hours during the week from 7:00 am PST to 4:00 pm PST, where the average acceptance rate is much lower at 21.2%.

Conclusions

When we set out to connect the productivity benefit of GitHub Copilot to usage measurements from developer

activity, we collected measurements

about acceptance of completions in

line with prior work, but also developed persistence metrics, which arguably capture sustained and direct

impact on the resulting code. We were surprised to find acceptance rate (number of acceptances normalized by the number of shown completions) to be better correlated with reported productivity than our measures of persistence.

In hindsight, this makes sense.

Coding is not typing, and GitHub Copilot's central value lies not in being the way users enter most of their code.

Instead, it lies in helping users to make the best progress toward their goals. A suggestion that serves as a useful template to tinker with may be as good or better than a perfectly correct (but obvious) line of code that only saves the user a few keystrokes.

This suggests that a narrow focus on the correctness of suggestions would not tell the whole story for these kinds of tooling. Instead, one could view code suggestions inside an IDE to be more akin to a conversation. While chatbots such as ChatGPT are already used for programming tasks, they are explicitly structured as conversations. Here, we hypothesize that interactions with Copilot, which is not a chatbot, share many characteristics with natural-language conversations.

We see anecdotal evidence of this in comments posted about GitHub Copilot online (see Appendix E for examples), in which users talk about sequences of interactions. A conversation turn in this context consists of the prompt in the completion request and the reply as the completion itself.

The developer's response to the completion arises from the subsequent changes incorporated in the next prompt to the model. There are clear programming parallels to factors such as specificity and repetition that have been identified to affect human judgements of conversation quality.¹⁸ Researchers have already investigated

the benefits of natural-language feedback to guide program synthesis,² so the conversational framing of coding completions is not a radical proposal.

But neither is it one we have seen followed yet.

--- END PAPER TEXT ---

Measuring GitHub Copilot's Impact on Productivity

Method: Claim Evidence Extraction

Prompt B:

You are extracting claims and evidence from a research paper about AI-assisted software engineering.

Produce a table with exactly five rows and the following columns:

Claim	Evidence Snippet	Citation (source_id, chunk_id)

Constraints / Guardrails

Each claim must be directly supported by the provided evidence snippet.

Evidence snippets must be copied verbatim or closely paraphrased from the text.

Citations must use the format: (source_id, chunk_id).

Do not invent claims, evidence, or citations.

If the paper does not contain enough information to extract five well-supported claims, include fewer claims and explicitly state “Insufficient evidence for additional claims.”

Do not rely on external knowledge.

Paper: Measuring GitHub Copilot's Impact on Productivity

--- BEGIN PAPER TEXT ---

CODE-COMPLETION SYSTEMS OFFERING suggestions to a developer in their integrated development environment (IDE) have become the most frequently used kind of programmer assistance.¹ When generating whole snippets of code, they typically use a large language model (LLM) to predict what the user might type next (the completion) from the context of what they are working on at the moment (the prompt).² This system allows for completions at any position in the code, often spanning multiple lines at once. Potential benefits of generating large sections of code automatically are huge, but evaluating these systems is challenging. Offline evaluation, where the system is shown a partial snippet of code and then asked to complete it, is difficult not least because for longer completions there are many acceptable alternatives and

no straightforward mechanism for labeling them automatically.⁵

An additional step taken by some researchers^{3,21,29} is to use online evaluation and track the frequency of real users accepting suggestions, assuming that the more contributions a system makes to the developer's code, the higher its benefit. The validity of this assumption is not obvious when considering issues such as whether two short completions are more valuable than one long one, or whether reviewing suggestions can be detrimental to programming flow.

Code completion in IDEs using language models was first proposed in Hindle et al.,⁹

and today neural synthesis tools such as GitHub Copilot, CodeWhisperer, and TabNine suggest code snippets within an IDE with the explicitly stated intention to increase a user's productivity. Developer productivity has many aspects, and a recent study has shown that tools like these are helpful in ways that are only partially reflected by measures such as completion times for standardized tasks.^{23,a} Alternatively, we can leverage the developers themselves as expert assessors of their own productivity.

This meshes well with current thinking in software engineering research suggesting measuring productivity on multiple dimensions and using self-reported data.⁶

Thus, we focus on studying perceived productivity.

Here, we investigate whether usage measurements of developer interactions with GitHub Copilot can predict perceived productivity as reported by developers. We analyze 2,631 survey responses from developers using GitHub Copilot and match their responses to measurements collected from the IDE. We consider acceptance counts and more detailed measures of contribution, such as the amount of code contributed by GitHub Copilot and persistence of accepted completions in the code. We find that acceptance rate of shown suggestions is a better predictor of perceived productivity than the alternative measures. We also find that acceptance rate varies significantly over our developer population as well as over time, and

present a deeper dive into some of these variations.

Our results support the principle that acceptance rate can be used for coarse-grained monitoring of the performance of a neural code synthesis system. This ratio of shown suggestions being accepted correlates better than more detailed measures of contribution. However, other approaches remain necessary for fine-grained investigation due to the many human factors involved.

Background

Offline evaluation of code completion

can have shortcomings even in tractable circumstances where completions can be labeled for correctness. For example, a study of 15,000 completions by 66 developers in Visual Studio found significant differences between synthetic benchmarks used for model evaluation

and real-world usage.⁷

The evaluation of context-aware API completion for Visual Studio IntelliCode considered Recall@5—the proportion of completions for which the correct method call was in the top five suggestions. This metric fell from 90% in offline evaluation to 70% when used online.²¹

Due to the diversity of potential solutions to a multi-line completion task, researchers have used software testing to evaluate the behavior of completions. Competitive programming sites have been used as a source of such data^{8,11} as well as handwritten programming problems.⁵

Yet, it is unclear how well performance on programming competition data generalizes to interactive development in an IDE.

In this work, we define acceptance rate as the fraction of completions shown to the developer that are subsequently accepted for inclusion in the source file. The IntelliCode Compose system uses the term click through rate (CTR) for this and reports a value of 10% in online trials.²⁰ An alternative measure is that of daily completions accepted per user (DCPU) for which a value of around 20 has been reported.^{3,29} To calculate acceptance rate

one must, of course, normalize DCPU by the time spent coding each day. For context, in our study, GitHub Copilot has an acceptance rate of 27% and a mean DCPU in excess of 312 (See Figure 1).^b These differences are presumably due to differences in the kinds of completion offered, or perhaps to user-interface choices. We discuss later how developer objectives, choice of programming language, and even time of day seem to affect our data.

Such discrepancies highlight the difficulty in using acceptance rate to understand the value of a system.

There is some evidence that acceptance rate (and indeed correctness) might not tell the whole story. One survey of developers considered the use of AI to support translation between programming languages and found indications that developers tolerated, and in some cases valued, erroneous suggestions from the model.²⁶

Measuring developer productivity through activity counts over time (a typical definition of productivity borrowed from economics) disregards the complexity of software development, as they account for only a subset of developer outputs. A more holistic picture is formed by measuring perceived productivity through self-reported data across various dimensions⁶

and

supplementing it with automatically measured data.⁴

We used the SPACE framework⁶

to design a survey that captures self-reported productivity and paired the self-reported data with usage telemetry.

To the best of our knowledge, this is the first study of code suggestion tools establishing a clear link between usage measurements and developer productivity or happiness. A previous study comparing GitHub Copilot against IntelliCode with 25 participants found no significant correlation between task completion times and survey responses.²²

Data and Methodology

Usage measurements. GitHub Copilot provides code completions using OpenAI language models. It runs within the IDE and at appropriate points sends a completion request to a cloudhosted instance of the neural model. GitHub Copilot can generate completions at arbitrary points in code rather than, for example, only being triggered when a developer types a period for invoking a method on an object. A variety of rules determine when to request a completion, when to abandon requests if the developer has moved on before the model is ready with a completion, and how much of the response from the model to surface as a completion.

As stated in our terms of usage,^b the GitHub Copilot IDE extension records the events shown in Table 1 for all users. We make usage measurements for each developer by counting those events.

Our measures of persistence go further than existing work, which stops at acceptance. The intuition here is that a completion which is accepted into the source file but then subsequently turns out to be incorrect can be considered to have wasted developer time both in reviewing it and then having to go back and delete it. We also record mostly unchanged completions: A large completion requiring a few edits might still be a positive contribution. It is not clear how long after acceptance one should confirm persistence, so we consider a range of options.

The events pertaining to completions form a funnel which we show quantitatively in Table 1. We include a summary of all data in Appendix A.c (All appendices for this article can be found online at <https://dl.acm.org/doi/10.1145/3633453>).

We normalize these measures against each other and write X_per_Y to indicate we have normalized metric X by metric Y. For example: accepted_per_hour is calculated as the total number of accepted events divided by the total number of (active) hour events.

Table 2 defines the core set of metrics we feel have a natural interpretation in this context. We note there are

alternatives, and we incorporate these in our discussion where relevant.

Productivity survey. To understand users' experience with GitHub Copilot, we emailed a link to an online survey to 17,420 users. These were participants of the unpaid technical preview using GitHub Copilot with their everyday programming tasks.

The only selection criterion was having previously opted in to receive communications. A vast majority of survey users (more than 80%) filled out the survey within the first two days, on or before February 12, 2022. We therefore focus on data from the four-week period leading up to this point ("the study period"). We received a total of 2,047 responses we could match to usage data from the study period, the earliest on Feb. 10, 2022 and the latest on Mar. 6, 2022.

The survey contained multiplechoice questions regarding demographic information (see Figure 2)

and Likert-style questions about different aspects of productivity, which were randomized in their order of appearance to the user. Figure 2 shows the demographic composition of our respondents. We note the significant proportion of professional programmers who responded.

The SPACE framework⁶ defines five dimensions of productivity: Satisfaction and well-being, Performance, Activity, Communication and collaboration, and Efficiency and flow. We use four of these (S,P,C,E), since self reporting on (A) is generally considered inferior to direct measurement. We included 11 statements covering these four dimensions in addition to a single statement: "I am more productive when using GitHub Copilot." For each self-reported productivity measure, we encoded its five ordinal response values to numeric labels (1 = Strongly Disagree, ..., 5 = Strongly Agree). We include the full list of questions and their coding to the SPACE framework in Appendix C. For more information

on the SPACE framework and how the empirical software engineering community has been discussing developer productivity, please see the following section.

Early in our analysis, we found that the usage metrics we describe in the Usage Measurements section corresponded similarly to each of the measured dimensions of productivity, and in turn these dimensions were highly correlated to each other (Figure 3). We therefore added an aggregate productivity score calculated as the mean of all 12 individual measures (excluding skipped questions). This serves as a rough proxy for the much more complex concept of productivity, facilitating recognition of overall trends, which may be less discernible on individual variables due to higher statistical variation. The full dataset of these aggregate productivity scores together with the usage measurements considered in this article is available at <https://bit.ly/47HVjAM>.

Given it has been impossible to produce a unified definition or metric(s) for developer productivity, there have been attempts to synthesize the factors that impact productivity to describe it holistically, include various relevant factors, and treat developer productivity as a composite measure^{17,19,24}. In addition, organizations often use their own multidimensional frameworks to operationalize productivity, which reflects their engineering goals—for example, Google uses the QUANTS framework, with five components of productivity.²⁷ In this article, we use the SPACE framework,⁶

which builds on synthesis of extensive and diverse literature by expert researchers and practitioners in the area of developer productivity. SPACE is an acronym of the five dimensions of productivity:

, S (Satisfaction and well being):

This dimension is meant to reflect developers' fulfillment with the work they do and the tools they use, as well as how healthy and happy they are with the work they do. This dimension reflects some of the easy-to-overlook trade-offs involved when looking exclusively at velocity acceleration—for

example, when we target faster turnaround of code reviews without considering workload impact or burnout for developers.

, P (Performance): This dimension aims to quantify outcomes rather than output. Example metrics that capture performance relate to quality and reliability, as well as further-removed metrics such as customer adoption or satisfaction.

, A (Activity): This is the count of outputs—for example, the number of pull requests closed by a developer. As a result, this dimension is best quantified via system data. Given the variety of developers' activities as part of their work, it is important that the activity dimension accounts for more than coding activity—for instance, writing documentation, creating design specs, and so on.

, C (Communication and collaboration): This dimension aims to capture that modern software development happens in teams and is, therefore, impacted by the discoverability of documentation or the speed of answering questions, or the onboarding time and processing of new team members.

, E (Efficiency and flow): This dimension reflects the ability to complete work or make progress with little interruption or delay. It is important to note that delays and interruptions can be caused either by systems or humans, and it is best to monitor both self-reported and observed measurements—for example, use self-reports of the ability to do uninterrupted work, as well as measure wait time in engineering systems).

What Drives Perceived Productivity?

To examine the relationship between objective measurements of user behavior and self-reported perceptions of productivity, we used our set of core usage measurements (Table 2). We then calculated Pearson's R correlation coefficient and the corresponding p-value of the F-statistic between each pair of usage measurement and perceived productivity metric. We also computed a PLS regression from all usage measurements jointly.

We summarize these results in

Figure 3, showing the correlation coefficients between all measures and

survey questions. The full table of all results is included in Appendix B, available online.

We find acceptance rate (accepted _ per _ shown) most positively predicts users' perception of productivity, although, given the confounding and human factors, there is still notable unexplained variance.

Of all usage measurements, acceptance rate correlates best with aggregate productivity ($p = 0.24$, $P < 0.0001$).

This measurement is also the best performing for at least one survey question in each of the SPACE dimensions.

This correlation is high confidence but leaves considerable unexplained variance. Later, we explore improvements from combining multiple usage measurements together.

Looking at the more detailed metrics around persistence, we see that it is generally better over shorter time periods than over longer periods. This is intuitive in the sense that shorter periods move the measure closer to acceptance rate. We also expect that

at some point after accepting the completion it becomes simply part of the code, so any changes (or not) after that point will not be attributed to GitHub Copilot. All persistence measures were less well correlated than acceptance rate.

To assess the different metrics in a single model, we ran a regression using projection on latent structures (PLS). The choice of PLS, which captures the common variation of these variables as is linearly connected to the aggregate productivity,²⁸ is due to the high collinearity of the single metrics. The first component, to which every metric under consideration contributes positively, explains 43 . 2% of the variance. The second component captures the acceptance rate/change rate dichotomy; it explains a further 13 . 1%. Both draw most strongly from acceptance rate.

This strongly points to acceptance rate being the most immediate indicator of perceived productivity, although it is beneficial to combine with others to get a fuller picture.

Experience

To understand how different types of

developers interact with Copilot, our survey asked respondents to self-report their level of experience in two ways:

- , "Think of the language you have used the most with Copilot. How proficient are you in that language?" with options 'Beginner', 'Intermediate', and 'Advanced'.
- , "Which best describes your programming experience?" with options starting with "Student" and ranging from "0–2 years" to "16+ years" in two year intervals.

We compute correlations with productivity metrics for both experience variables and include these two variables as covariates in a multivariate regression analysis. We find that both are negatively correlated with our aggregate productivity measure (proficiency: $\rho = -0.095$, $P = 0.0001$; years of experience: $\rho = -0.161$, $P < 0.0001$). However, in multivariate regressions predicting productivity from usage metrics while controlling for demographics, proficiency had a non-significant positive effect (coeff = 0.021, $P = 0.213$), while years of experience had a non-significant negative effect (coeff = -0.032 , $P = 0.122$). Looking further at individual measures of productivity, (Table 3) we find that both language proficiency and years of experience negatively predict developers agreeing that Copilot helps them write better code. However, proficiency positively predicts developers agreeing that Copilot helps them stay in the flow, focus on more satisfying work, spend less effort on repetitive tasks, and perform repetitive tasks faster. Years of experience negatively predicts developers feeling less frustrated in coding sessions and performing repetitive tasks faster while using Copilot, but positively predicts developers making progress faster when working in an unfamiliar language. These findings suggest that experienced developers who are already highly skilled are less likely to write better code with Copilot, but

Copilot can assist their productivity in other ways, particularly when engaging with new areas and automating routine work.

Junior developers not only report higher productivity gains; they also tend to accept more suggestions. However, the connection observed in the section "What Drives Perceived Productivity" is not solely due to differing experience levels. In fact, the connection persists in every single experience group, as shown in Figure 5.

Variation over Time

Its connection to perceived productivity motivates a closer look at the acceptance rate and what factors influence

it. Acceptance rate typically increases over the board when the model or underlying prompt-crafting techniques are improved. But even if these conditions are held constant (the study period did not see changes to either), there are more fine-grained temporal patterns emerging.

For coherence of the cultural implications of time of day and weekdays, all data in this section was restricted to users from the U.S. (whether in the survey or not). We used the same time frame as for the investigation in the previous section. In the absence of more fine-grained geolocation, we used the same time zone to interpret timestamps and for day boundaries (PST), recognizing this will introduce some level of noise due to the inhomogeneity of U.S. time zones.

Nevertheless, we observe strong regular patterns in overall acceptance rate (Figure 6). These lead us to distinguish three different time regimes, all of which are statistically significantly distinct at $p < 0.001\%$ (using bootstrap resampling):

- , The weekend: Saturdays and Sundays, where the average acceptance rate is comparatively high at 23.5%.

- , Typical non-working hours during the week: evenings after 4:00 pm PST until mornings 7:00 am PST, where the average acceptance rate is also rather high at 23%.

- , Typical working hours during the week from 7:00 am PST to 4:00 pm PST, where the average acceptance rate is

much lower at 21.2%.

Conclusions

When we set out to connect the productivity benefit of GitHub Copilot to usage measurements from developer activity, we collected measurements about acceptance of completions in line with prior work, but also developed persistence metrics, which arguably capture sustained and direct impact on the resulting code. We were surprised to find acceptance rate (number of acceptances normalized by the number of shown completions) to be better correlated with reported productivity than our measures of persistence.

In hindsight, this makes sense.

Coding is not typing, and GitHub Copilot's central value lies not in being the way users enter most of their code.

Instead, it lies in helping users to make the best progress toward their goals. A suggestion that serves as a useful template to tinker with may be as good or better than a perfectly correct (but obvious) line of code that only saves the user a few keystrokes.

This suggests that a narrow focus on the correctness of suggestions would not tell the whole story for these kinds of tooling. Instead, one could view code suggestions inside an IDE to be more akin to a conversation. While chatbots such as ChatGPT are already used for programming tasks, they are explicitly structured as conversations. Here, we hypothesize that interactions with Copilot, which is not a chatbot, share many characteristics with natural-language conversations.

We see anecdotal evidence of this in comments posted about GitHub Copilot online (see Appendix E for examples), in which users talk about sequences of interactions. A conversation turn in this context consists of the prompt in the completion request and the reply as the completion itself.

The developer's response to the completion arises from the subsequent changes incorporated in the next

prompt to the model. There are clear programming parallels to factors such as specificity and repetition that have been identified to affect human judgements of conversation quality.¹⁸ Researchers have already investigated the benefits of natural-language feedback to guide program synthesis,² so the conversational framing of coding completions is not a radical proposal. But neither is it one we have seen followed yet.

--- END PAPER TEXT ---