More Code, Less Understanding? On the Impact of AI Assistants on Developers' Productivity and Code Ownership

Alberto Martin-Lopez,* Rosalia Tufano,* Emanuela Guglielmi,* Ana B. Sánchez,* Ana Lourdes Sanz, Simone Scalabrino, Rocco Oliveto, Sergio Segura, and Gabriele Bavota

***Abstract*—Artificial Intelligence (AI) is transforming many domains, including software engineering. AI-based tools are gaining popularity and are increasingly being integrated into software development workflows, automating complex tasks such as code writing and reviewing. When it comes to coding tasks, some evidence suggests that tools like Copilot boost developers' productivity (*e.g.,* developers can handle a larger number of pull requests per week). However, it remains unclear whether this comes at the expense of code ownership (*i.e.,* the developers' ability to argue about their implementation choices). To partially address this gap, we present an experiment aimed at investigating the impact of AI-based assistants on developers' productivity and behavior in the context of code writing (*e.g.,* developing a program from scratch or evolving an existing code). Our focus is on the interplay between productivity and code ownership. We asked 69 participants (34 BSc and 13 MSc students, 8 researchers, and 14 professional developers) to perform two code writing tasks, one with the support of AI and one without. Then, we compared the two treatments in terms of: (i) time spent on the coding task and percentage of the task completeness—both being productivity proxies; and (ii) ability of the participants to answer questions about the code they implemented—code ownership proxy. While the time-based analyses did not provide strong evidence on the impact of AI on the investigated dependent variables, participants using AI achieved a much higher task completeness (>2× in terms of median), confirming a positive impact on their productivity. However, such a boost did not come for free. Indeed, we also observed a loss in code ownership when participants used AI, with lower ability to answer technical questions (–12.5%).**

***Index Terms*—AI Assistants, Developer Productivity, Code Ownership, Controlled Experiment.**

I. Introduction

Artificial Intelligence (AI) is reshaping the way in which software is built and maintained. AI-based tools such as ChatGPT [1] and Copilot [2] are increasingly integrated into development workflows, assisting with tasks like code generation and review [3]. Empirical studies demonstrated that these tools can help developers in becoming more productive, for example, by finalizing more implementation tasks per week [4]. However, given their recent emergence, little is known about possible side effects of their adoption. For instance, while a developer may implement a requested feature faster using Copilot, it is unknown whether this may result in a loss of "code ownership," namely their ability to understand[1] and explain their implementation choices and be accountable for the produced code. This concept is particularly crucial in large, collaborative projects where multiple developers contribute to the same codebase [5], [6]. Recent findings at MIT further support this concern, showing that users of LLMs reported a significantly lower sense of ownership over their written work compared to those using search engines or no tools at all [7]. While most of the research in AI for software engineering focuses on presenting novel techniques exploiting Deep Learning (DL) for the automation of software-related tasks [8], [9], we argue that there is a need to at least partially shift the focus toward *building empirical evidence on how AI-based assistants affect developers' effectiveness and behavior*.

We take a step in this direction by presenting the findings of a family of experiments assessing the impact of AI-based tools on the developers' productivity and code ownership. We involved in our study 69 participants, being 34 BSc, 13 MSc Computer Science students, and 8 Software Engineering researchers (6 PhD students and 2 postdocs) from three different Universities (U1, U2, U3), and 14 professional developers working in technological companies. Each participant was asked to perform two coding tasks, one with the support of AI (WAI) and one without (NAI). The two tasks concerned either the development from scratch of code implementing given requirements, or the evolution of an existing code to meet specific change requests. Therefore, our experiment follows a crossover design with the treatment and the task type as factors. In total, we ran our experiment six times.[2] For instance, BSc and MSc students from U1 did the experiment in the same *run*, while professional developers were split across different runs as they were located in different countries. While all runs adopted the same experimental design, we crafted different tasks which were appropriate for the respective participants in terms of complexity, domain, and programming language to

Alberto Martin-Lopez, Ana B. Sánchez and Sergio Segura are with the SCORE Lab, I3US Institute, Universidad de Sevilla. Rosalia Tufano and Gabriele Bavota are with the Università della Svizzera italiana. Emanuela Guglielmi, Simone Scalabrino and Rocco Oliveto are with the Università degli Studi del Molise. Ana Lourdes Sanz is with Schneider Electric.

*These authors led the work in each of the countries they were located in at the time of the study: Alberto Martin-Lopez and Rosalia Tufano in Switzerland; Emanuela Guglielmi in Italy; and Ana B. Sánchez in Spain.

[1]We use the term "understanding" to refer to developers' ability to reason about, explain, and recall aspects of code they have recently implemented. This notion is closely related to developers' experiential familiarity and sense of ownership over the code, rather than to the intrinsic readability or comprehensibility of the code itself.

[2]A *run* refers to a single execution characterized by the same physical location and the same person supervising the experiment.

be used. Similarly, the complexity of the tasks was different across the categories of participants (those for researchers and professional developers were the most complex).

In the WAI treatment (*i.e.,* AI-based tools allowed), participants were free to access tools like ChatGPT or Copilot which were instead forbidden in the other treatment (NAI). Participants had a maximum of 75 minutes per task. The experiments were all run in a controlled setting, with participants violating the instructions received being excluded from the study.

We assess and compare two aspects among the two treatments: *productivity* and *code ownership*. As for the former, we use as primary proxy the *completeness* of the coding task, manually assessed via code inspection. On top of it, we also measure the *completion time* for the task, relevant when participants did not use the whole 75 minutes available. As for code ownership, we assess the ability of participants to answer questions about the code they implemented. This was the most challenging part of our study. Indeed, running hundreds of one-to-one interviews (in total, 180 coding tasks were performed by 90 involved participants) was not an option due to the excessive cost in terms of time, both on the experimenters' and on the participants' side. We address this challenge with a different approach: we developed an infrastructure that, at the end of each coding task, triggers the OpenAI `o1` reasoning model [10] to automatically generate several technical questions about the implemented code, and asked participants a written answer to these questions. As primary proxy for code ownership we measure the percentage of correctly-answered questions. Also, we track the time required to provide an answer, with the conjecture that participants in the WAI group may require longer due to the need for comprehending the AI-generated code.

Our main findings show that:

1) *AI support boosts productivity.* We observed a major boost in productivity for participants subjected to the WAI treatment, with task completeness being twice as high as compared to participants in the NAI group (based on the median scores). This confirms previous findings coming from field studies [4].
2) *There is a price to pay in terms of code ownership.* Participants supported by AI struggled more in answering the technical questions about the code they implemented. Indeed, the median of correct answers for participants using AI is 87.5%, as compared to the 100% of participants who wrote code without AI support.

Study material and data are publicly available [11].

II. Study Design

The *goal* of our study is to assess the impact of AI-based tools on the productivity and code ownership of developers during implementation tasks. The *context* is represented by 69 participants, each performing one coding task with and one without the support of the AI, and eight coding tasks, either in Java or Python. The study addresses the following research questions (RQs):

RQ$_1$: *To what extent does AI-based support impact the productivity of developers during coding tasks?* We compare the task completeness achieved by participants with/without the usage of AI and the time needed to finalize the task.

TABLE I
PARTICIPANTS INVOLVED IN OUR STUDY.

University/Company	BSc	MSc	Researcher	Professional	Total
U1	14	1	3	-	**18**
U2	20	12	1	-	**33**
U3	0	0	4	-	**4**
C	-	-	-	14	**14**
Total	**34**	**13**	**8**	**14**	**69**

RQ$_2$: *To what extent does the availability of AI-based support during code writing impact the developer's code ownership?* At the end of each coding task we ran a questionnaire aimed at assessing the participants' understanding of the code they developed, to assess whether there is a loss of ownership when AI-based tools were used as support during development. In this paper, we use code ownership in a narrower sense than is common in software engineering, where it often refers to long-term familiarity or responsibility over code. Here, we operationalize it through participants' ability to answer questions about code they had just implemented, and therefore treat it as a short-term, comprehension-based proxy for code ownership.

A. Context Selection

1) Participants: For our study we targeted participants having different levels of coding experience, since we want to observe whether such an independent variable has an impact on the dependent variables. It is possible, for example, that a loss of code ownership may be observed only for inexperienced developers, while not for professionals having wide coding expertise. We used convenience sampling [12], [13] to recruit 90 participants who are either BSc/MSc students in Computer Science, researchers in Software Engineering, or professional developers in a technological company. In total, we involved 42 BSc and 21 MSc students, 9 researchers, and 18 professional developers.[3] Out of those, we had to exclude 21 of them for reasons we will document after explaining the experimental design. At the end, we collected valid data from 69 participants, being 34 BSc and 13 MSc students, 8 researchers, and 14 professional developers. We only involved BSc students who successfully passed at least one programming course. Table I summarizes the provenance of participants, with U1, U2, and U3 indicating the three universities in which we run the study,[4] and C indicating technological companies in which the professional developers work (four different, overall). The study was conducted across three different countries (Italy, Spain, and Switzerland).

We asked participants to complete a questionnaire in which they indicated their: (i) highest level of education completed; (ii) current position; and (iii) years of programming experience. Also, we asked whether they used AI-based tools in the past in the context of software-related tasks. Fig. 1 depicts

[3]All of them provided informed consent to participate in the study.

[4]U1: Università degli Studi del Molise (Italy); U2: Universidad de Sevilla (Spain), U3: Università della Svizzera italiana (Switzerland).

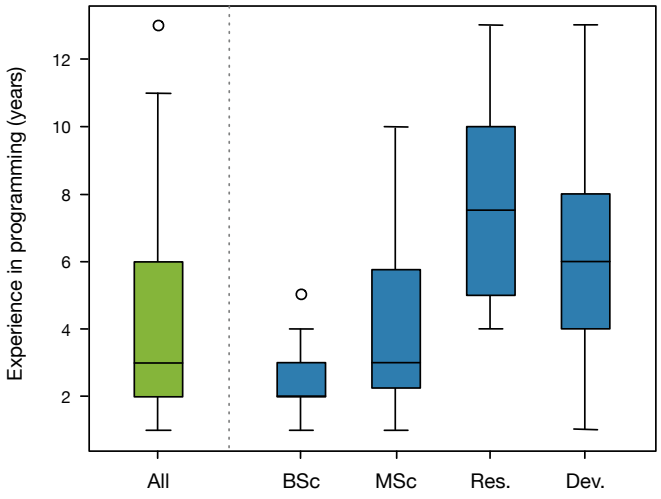

Fig. 1. Participants programming experience.

the years of programming experience for each "category" of participants (blue boxes) and for the overall sample (green). The collected data will be used as possible confounding factors in our data analysis.

2) Coding Tasks: We asked each participant to perform two coding tasks: one with the support of AI and one without. Among these two tasks, one was a greenfield development task and one an evolution task on a given code.

All researchers and professionals performed the same Python tasks, regardless of their university or company. As for the BSc and MSc students, which came from both U1 and U2, the tasks differed to better suit students' knowledge. Table II summarizes the tasks performed during the study, including their type—development (tasks starting with letter 'D') or evolution (starting with letter 'E')—the programming language, the organizations and participants involved, and a short description. Next, we summarize each task. Note that some of the task descriptions may look ambiguous due to details omitted for the sake of brevity. The detailed tasks' descriptions are available in our replication package [11].

Task D1. Develop a command-line application in Python that manages a gift wish list. Users should be able to add, view, update, and remove gift ideas for different people. The application should save the wish list to a CSV file when exiting and load it when starting, preserving the data between sessions.

Task E1. You are provided with a simple Python implementation of the classic Snake game (113 LOC). Your task is to add four features to the provided implementation: (i) The current implementation does not keep track of the user's score; implement the logic needed to increase the score for each piece of food eaten. (ii) Currently, the player loses if the snake collides with one of the four walls; remove such a behavior, and implement a wrap-around logic. (iii) Currently, the game immediately starts when `main.py` is launched; modify such a behavior so that the user is asked for a non-empty nickname. (iv) When the user loses, its score must be saved in a CSV file (`scores.csv`) assuming it is one of the top-10 scores of all time.

Task D2. Develop a command-line application in Java which allows university students to manage their exam planning. Users should be able to add, view, modify, and mark exams as completed. The program must provide a simple overview showing for which courses the preparation of the corresponding exam is ahead of schedule, on schedule, and behind schedule, based on rules specified in the detailed requirements attached to the task.

Task E2. You are provided with a Java implementation of the Tic Tac Toe game (112 LOC). Your task is to add several features. Currently, the game immediately starts when launching the program. Create a menu which allows to start a game, to set the nickname of the players, and to visualize a report concerning all played games organized by player. Such report features, for each player, the number of games played, the average duration of the game, and the number of wins, ties, and losses. Additional requested features concern the possibility to reset the game statistics shown in the report and the visualization on screen of the score for the current game.

Task D3. Develop an HTTP client for the GitLab API [14] in Java. Given the ID of a GitLab project and n requested results, implement these operations: (i) Get the first n issues of the project. (ii) Get the most voted issue of the project among the first n; if more than one, return all of them. (iii) Get the distinct author names of the first n issues of the project. (iv) Get the issues (closed or not) opened for more than one year among the first n.

Task E3. You are provided with a Spring Boot project [15] implementing a REST API [16] for handling movies (154 LOC). You are requested to: (i) replace `for` loops with Java Streams; (ii) throw an `IllegalStateException` exception if any movie found has a `null` or empty title; (iii) implement a method that, given a list of movies, returns them ordered by descending order of year of release; and (iv) load the movies from an `XML` file instead of a `JSON` file.

Task D4. Given a movie manager written in Java (213 LOC) create a test suite for it from scratch, making effective use of JUnit 5 features [17]. The test suite should implement a setup method reused in all tests. Add tests for the features related to adding a movie and searching by genre. Use parameterized tests with data provided from CSVs and external methods to test the features related to searching by title and by rating range, including JUnit assumptions to skip tests if not enough data is provided. Use mocks to test the behavior related to empty data and duplicated movies.

Task E4. You are provided with both the implementation and the test suite of an HTTP client for the GitLab API written in Java (959 LOC overall). The current test suite invokes the GitLab API. You must adapt it to use mocks with fake data, decoupling the tests from the actual web service. Furthermore, the implementation of the HTTP client contains four bugs. You must identify these, modify the current tests to catch them, and then fix the implementation.

A couple of observations are worth being made concerning the above-summarized tasks. First, we designed them to cover application domains the participants are familiar with: For researchers and professional developers we targeted programs

TABLE II
SUMMARY OF THE TASKS CONDUCTED IN THE STUDY.

ID	Type	Language	Organizations	Participants	Short description
D1	Development	Python	U1, U2, U3, C	Researchers and professionals	Implementing a gift list application
E1	Evolution	Python	U1, U2, U3, C	Researchers and professionals	Extending the Snake game with new features
D2	Development	Java	U1	BSc and MSc students	Implementing an exam planning application
E2	Evolution	Java	U1	BSc and MSc students	Adding extra features to Tic Tac Toe game
D3	Development	Java	U2	BSc students	Implementing a GitLab API client
E3	Evolution	Java	U2	BSc students	Improving a movie REST API
D4	Development	Java	U2	MSc students	Creating a JUnit 5 test suite for a movie manager
E4	Evolution	Java	U2	MSc students	Adding test mocks and bug fixes to a GitLab client

that we believed any seasoned developer would understand (*i.e.,* wish list and snake game). Besides, we deliberately chose popular domains on which LLMs work reasonably well. Indeed, the focus of our study is really on the interplay between productivity and code ownership and, to study that, we need tasks for which modern LLMs can actually help developers writing code, to then see whether there was a price to pay in terms of code ownership. When it comes to BSc and MSc students, the tasks were designed considering: (i) their programming knowledge, especially in terms of programming language they studied in the past; and (ii) the context of the course within which the experiment was performed. As per the students coming from U1, both BSc and MSc students performed the same tasks (D2 and E2 above), since they both already passed a Java course. While MSc students are expected to be more proficient, BSc students studied Java in the previous semester (Fall 2024), thus having their knowledge of such a programming language quite fresh.

As per the BSc and MSc students from U2, we opted for different tasks (D3/E3 and D4/E4). This is because, when we ran the experiment, the BSc students were finishing a course on Software Architecture and Integration, with a focus on the development and consumption of REST APIs. Therefore, we designed the tasks to be related with these topics, aiming to increase their complexity while keeping them manageable for the students' level. MSc students, instead, were finishing a course on Advanced Software Testing, leading to the design of tasks related to test code.

B. Treatments, Setup and Procedure

Our study follows a within-subject crossover design, with two factors: whether AI assistants are allowed or not, and the type of coding task to perform. This means that each participant experiences both conditions, working on different tasks with and without AI. Regarding the first factor, we consider two treatments. The first, named *with AI support* (WAI), allows participants to access any AI-based tools they want. The second, named *no AI support* (NAI), forbids participants from using AI-based tools. In both treatments, participants had access to any online resource they wanted (besides AI-based tools) to better simulate a real development scenario (*e.g.,* they could access discussions on Stack Overflow). Regarding the second factor, we consider tasks in Table II: development (TD) and evolution (TE). In practice, we aim to investigate whether our findings in terms of developer productivity and code ownership when using or not using AI-based tools are consistent across different task types.

TABLE III
EXPERIMENTAL DESIGN.

Session	Group A	Group B	Group C	Group D
1	TD-WAI	TE-WAI	TD-NAI	TE-NAI
2	TE-NAI	TD-NAI	TE-WAI	TD-WAI

The experimental design we adopted is a classical paired design for experiments with two factors of two possible values each, reported in Table III. As illustrated, this crossover design has two periods (sessions 1 and 2) and four sequences (groups A, B, C, and D). Each sequence represents a specific combination of the two factors considered in our study. During each run of our experiment, participants were randomly assigned to one of the four groups. Each participant performed two different coding tasks, one in session 1 and one in session 2. Such a design ensures the minimization of confounding factors such as: (i) tiring and learning effects, since half of participants start with the WAI treatment (groups A and B) and half with NAI (C and D); the effect of task complexity, since half of participants apply WAI to the development task (A and D) and half to the evolution task (B and C).

We started each of the six runs of our experiment with a brief presentation, explaining (i) what participants were expected to do and (ii) the environment used for the experiment. For the latter, we provided each participant with detailed instructions to connect to a server we set up using the Remote Development plugin of Visual Studio Code [18]. The server had the necessary environment configured for the study, depending on the experimental group the participant was assigned to.

For example, if the participant was assigned to group A, they would have GitHub Copilot installed and ready to run during the first session (*i.e.,* development task with AI), while the AI-based assistant was blocked in the second session. The correctness of the setting was double-checked for each participant by the experimenters present in the room, who also monitored the task execution to make sure that participants with the NAI treatment did not use AI-based tools from outside the IDE (*e.g.,* from the browser). We pre-installed Java (version 17) and Python (version 3.10) to support running any of the subject tasks. Additionally, we set up Visual Studio Code with the Java Extension Pack [19] and the Python extension [20]. Finally, we used the Tako VS Code plugin [21] to record all the actions performed in the IDE. Tako records events such as opening and closing files and tabs, switching between them and editing files, among others. This allows us

TABLE IV
EXAMPLES OF TECHNICAL QUESTIONS GENERATED BY o1.

Task	Type	Question
D1	Closed	*Which part checks if a numeric input is above a certain threshold?*
E1	Closed	*Where do you trigger the final data-handling routine once the game can no longer proceed?*
D2	Closed	*Where do you ensure that the study progress is within a valid range before it is assigned to the exam?*
E2	Closed	*Where in the newly added code do you compute the total duration of each game?*
D3	Open	*Describe in natural language all code blocks in the getOpenedIssueForMoreThanOneYear function.*
E3	Open	*Which approach did you use to compare the creation date of each issue with a specific time threshold, and why did you choose it?*
D4	Open	*If we removed the testSearchMoviesByMultipleGenresWithMismatchedFormat test, what kinds of problems in the searchByGenre method could potentially go undetected?*
E4	Open	*Imagine the method getMostVotedIssues is changed to ignore issues with upvotes below a certain threshold. How would you adjust your test suite to detect any regression caused by that change?*

to perform the time-based analyses needed to answer our RQs.

Once connected, participants could see the project related to the specific task, which was basically an empty folder in case of the development tasks, or a starting code base for the evolution ones. Each project included a PDF with instructions about what was expected for the specific task (see the replication package [11]). Participants had a maximum of 75 minutes to complete each task.

Once each task was completed, it was time to collect data to assess the code ownership of the participants. To this end, we initially considered conducting one-to-one interviews. However, we discarded this option due to the excessive cost in terms of time, both for experimenters and participants. For example, the largest run of our study involved 24 participants. This means that even assuming two experimenters running the technical interviews in parallel (12 each), the last participant should have waited for 11 interviews to be completed before running their interview, both at the end of the first and of the second task. Assuming an optimistic duration of 10 minutes per interview, this would have resulted in a total of four hours of interviews, two at the end of each task. To overcome this limitation, we performed *semi-automated offline technical interviews*: we used the OpenAI o1 reasoning model [10] to automatically generate technical questions about the implemented code, and asked participants to provide written answers to these questions (that, as we will explain, we still manually reviewed). This allowed the collection of individualized data from all participants in a significantly shorter time. To generate the questions, the implemented code was automatically saved on our server upon completion of each task. The experimenter then executed a Python script we developed using OpenAI APIs along with carefully crafted prompts.

The specific script used for each task is available as a part of the replication package [11]. The prompts were designed following established prompt engineering practices, including role-playing, structured formatting, and in-context learning [22]. First, we made it clear to the LLM its role for the required task:

> *You are a lecturer in Computer Science whose goal is to evaluate the source code of a program written by a student to implement a specific request described below (# Task). You must carefully analyze the student's code and ask questions to find out if they understand the code they have written.*

Then, we explicitly asked o1 to generate the technical questions. Here there is a distinction between the development and evolution task. In the former, we asked the LLM to generate questions concerning the entire code present in the IDE directory containing the developed code. For the evolution task, instead, we instructed the LLM to only ask questions about the *diff* between the original project given to the participants, and the final version implementing the required code changes.[5] This was needed to make sure that o1 was only asking questions about the code actually written by the participants, rather than on code we provided to them.

For the tasks D1, E1, D2, E2, D3, and E3 described above, we asked o1 to generate (i) six questions which can be answered by indicating the line number of the code performing a certain operation, and (ii) three open-ended questions, asking questions about the program logic, and the reasons behind the performed implementation choices. For tasks D4 and E4 which, as explained before, concerned test code rather than production code, we found out that closed questions (*i.e.,* those asking for the code performing a certain operation) were trivial to answer, since they were mostly focusing on assert statements (*e.g., where do you check that the number of returned products is the expected one?*). Thus, we asked the LLM to generate six open-ended questions for these two specific tasks. Table IV reports eight examples of generated questions (four closed and four open), one for each of the eight tasks. Note that while some questions may ask about the rationale for some implementation choices (*e.g.,* E3 example), others may concern understandability aspects (*e.g.,* E1 example). All in all, we evaluate code ownership taking into account both types of questions.

The prompts provided to o1 also featured several clarifications aimed at guiding it towards the generation of meaningful and non-trivial questions. For example, we clarified that it had to "*avoid providing explicit details in the questions such as literal strings or conditions, since then those would be easy to spot in the code*". Still, some of the generated questions were either meaningless or trivial, and were excluded from our study as documented in Section II-C.

[5]The *diff* was automatically generated by our script and sent to the OpenAI API as part of the prompt.

Once the questions were generated in a `txt` file placed in the IDE, we gave up to 20 minutes to the participants to answer the questions. Participants could not leave the IDE during this time: they were only allowed to look back at the code they wrote to answer the questions, without any support (neither AI nor other type, *e.g.,* Google). The total duration of each session of our experiment was 95 minutes at maximum (75 minutes for the implementation, 20 minutes for the question answering). After the first session, participants had a break before starting with the subsequent session. At the end of each run, we collected for both tasks of each participant: (i) the code written; (ii) the questions generated by `o1` and the provided answers; and (iii) the monitoring data collected by Tako.

Our replication package [11] features an "*Experiment reporting*" document specifying the exact setup of the LLM used in our study (*e.g.,* LLM's settings, prompts, etc.) as well as ethical considerations and data privacy.

C. Data Collection & Analysis

To address $\mathbf{RQ}_1$, one of the authors examined the output code submitted by each participant at the end of each task.[6] Using a predefined checklist (available in [11]), the reviewer assigned a "*completeness*" score to each submission. Each checklist outlined the specific sub-tasks required for a given task and the corresponding scores for each. For example, the checklist for task D1 includes the following items:

- ☐ Add a new gift idea with recipient's name, description, occasion, and price. [+10]
- ☐ Search gifts in a given price range. [+15]

The scores in brackets reflect the complexity of each sub-task, with more complex items earning higher points. The maximum possible completeness score for a task is 100, awarded when all sub-tasks are correctly implemented. Points were only assigned if a sub-task was fully functional (*e.g.,* in task D1, the gift search feature needed to work correctly when the code was executed). The correctness of a given sub-task was assessed by running the code (thus, observing that its behavior was aligned with the provided specification) and by inspecting the code to further verify the implemented logic. To increase our confidence in the assigned scores, a second author independently reviewed each task, re-examining the submitted code. Inter-rater reliability was quantified using the intraclass correlation coefficient (ICC), two-way random-effects model with absolute agreement. Inter-rater agreement was excellent (0.96), indicating high consistency between assessors. Disagreements were resolved through discussion prior to analysis.

Besides *completeness*, which represents our primary endpoint to answer RQ_1, we also consider as secondary endpoint the time participants took to finalize the tasks as automatically collected by the Tako plugin [21] (*implementation time*).

We answer RQ_1 by visually comparing the distributions of *completeness* and *implementation time* via boxplots for tasks performed with (WAI treatment) and without (NAI) the support of AI. In addition to that, we run the following statistical analyses. We use a mixed-effects linear model to assess the impact of treatment on *completeness* (dependent variable). The independent variable is the *treatment factor—i.e.,* use of AI (WAI and NAI)—while the control variables are: (i) highest level of education completed; (ii) current position (*e.g.,* BSc student, Professional); (iii) years of programming experience; (iv) programming language of the subject task (Java/Python); (v) specific task (D1, D2, D3, D4, E1, E2, E3, E4); and (vi) the session in which the task was performed (*i.e.,* first or second session). Independent variables being linearly dependent are not included when building the model to avoid potential issues of multicollinearity. While we considered also including the "past usage of AI-based tools for software-related tasks" as one of the control variables, all participants answered yes to this question, making its analysis irrelevant. Also, the model considers the specific experimental run and the participant as random effects. The former aims at separating the possible peculiar characteristics of a specific run of the experiment from the effect produced by the investigated factor, while the latter aims at capturing between-person heterogeneity.

We also build a log-normal mixed-effects model to assess the impact of the treatment on the *implementation time* (dependent variable). The choice to use a log scale to model *implementation time* is due to the fact that time outcomes are positive and right-skewed. The log-normal mixed-effects model enables better-behaved residuals and interpretable multiplicative effects (time ratios). Its independent variables and random effects are the same discussed for the mixed-effects linear model used to assess completeness. The only difference is the addition of "Lines of Code (LOC)" as further independent variable, to check whether longer implementation times might just be explained by more written code.

To answer $\mathbf{RQ}_2$, we manually inspected for each of the 138 completed tasks: (i) the questions generated by `o1`, and (ii) the answers provided by participants. Checking the questions was required to make sure that `o1` was actually asking meaningful and non-trivial questions. We considered a question "meaningful" if it was understandable, clearly related to the participant's code, and required reasoning about implementation choices or code behavior. Questions were discarded as non-meaningful if they referred to decisions that were dictated by task requirements rather than made by the participant (*e.g.,* asking why a specific feature was implemented when it was explicitly mandated by the task description). We considered a question trivial if its answer could be obtained through a simple keyword search in the code, without requiring comprehension of the implementation. For example, a question asking "*where is the price converted from String to double?*" was discarded if the code included an explicit comment such as "`converting price from String to double`". Each question was initially reviewed by one author and double-checked by a second author, following the same criteria. Disagreements were rare (8 out of 1,170 generated questions) and typically arose from subjective judgments about question understandability or whether a question required reasoning beyond surface-level code inspection. In such cases, we adopted a conservative approach and discarded the question if it was deemed unclear or borderline by at least one reviewer. All disagreements were

[6]This process involved five authors, each responsible for reviewing a portion of the 138 collected tasks.

resolved through open discussion. In total, 120 out of the 1,170 o1-generated questions were discarded. Across all tasks, we retained an average of 7.6 valid questions per task, with a minimum of 4 and a maximum of 9. Our replication package [11] reports the distribution of valid questions both per task and per cohort, showing no relevant differences in the number of questions for the WAI and NAI treatment.

As per the answers provided by participants, we thought about several grading schemes, opting for a boolean one (*i.e.,* wrong or correct) after inspecting the answers provided in the first run of our experiment. Indeed, given the types of questions asked by o1 (*i.e.,* mostly explanations about the written code), participants either answered correctly or not at all, with little room for partial correctness. For example, when the LLM was asking in which part of the code a specific behavior was implemented, participants either indicated the correct code or not. On top of that, adding more complex grading schemes would have increased the subjectivity in the assessment. Also in this case each answer was graded as correct or wrong by one author and double-checked by a second one, with 7 cases of disagreement solved via open discussion.

Using the collected grades, we compute for each participant and each task their *code ownership* level, as the percentage of correct answers they provided to the non-discarded questions. This is our primary endpoint to answer RQ_2. These distributions are compared between treatments via boxplots. To mirror the analyses performed in RQ_1, we also wanted to build a linear mixed model having *code ownership* as dependent variable. However, as explained by Bartlett and Partnoy [23], using ratio variables as dependent variables can cause spurious correlations, resulting in noise in the statistical analysis. For this reason, we define a boolean dependent variable named *"Is question correctly answered"* indicating whether each of the (non-discarded) questions has been correctly answered by the participant. This means that a participant who answered six questions will contribute with six data points to this analysis. Using this data, we build a logistic mixed-effects model having *"Is question correctly answered"* as dependent variable, the treatment factor as the independent variable, and seven confounding variables: The six already presented for RQ_1 (*i.e.,* highest level of education completed, current position, years of programming experience, programming language of the subject task, specific task, and the session in which the task was performed) plus the type of question each data point refers to (*i.e.,* open or closed). Indeed, as previously explained, our experiment features both open and closed questions. Also this model features the specific experimental run and the participant as random effects.

As a secondary endpoint for code ownership, we also analyze the time taken by participants to answer questions (*time to answer*), performing the same analyses described for *implementation time* in RQ_1, with one exception. Because participants received different numbers of valid ownership questions, we modeled answering time using a log-normal mixed-effects model with the number of valid questions included as an offset. This specification accounts for unequal exposure across participants by effectively modeling the average time per question, rather than total answering time, while retaining all observations in the analysis. Also, it is important to note that, differently from what happened for *implementation time* in which participants could leave the IDE (*e.g.,* to browse the web), while answering questions this was not allowed. Thus, the two time-related measurements should be interpreted differently.

1) Robustness Checks and Sensitivity Analyses: Our study follows a two-period crossover design, in which each participant performed two tasks, one in each session. Thus, we conducted a set of robustness checks to assess the potential impact of period and carryover effects on all outcome measures, including *task completeness* and *implementation time* (RQ_1), as well as *code ownership* and *time to answer* (RQ_2).

We first extended the primary specification by adding a Treatment $\times$ Session interaction term. The interaction term tests whether the treatment effect differs across the two sessions, which would be indicative of order-dependent or carryover effects. Also, we conducted a sensitivity analysis restricted to Session-1 observations only, where no carryover can occur by design. This analysis provides an unbiased parallel-group comparison between treatments and serves as a conservative check on the robustness of the main findings.

D. Excluded Participants

While we ran our study with 90 participants, we had to exclude 21 of them due to different reasons. Five of them did not complete the experiment. Eight did not manage to write enough code so that o1 could generate the required questions (*e.g.,* one only wrote five lines of code). In three cases the code developed by participants was not sent to o1 for the questions generation due to the fact that they did not follow our instructions, implementing their code in a new project they created in the IDE rather than in the predefined project. Three did not answer any questions and, from the inspection of the logs collected via Tako [21], we found they never opened the file with the o1's questions. Finally, two used AI when they were not supposed to, and thus were excluded from the study. The 69 left participants are still quite balanced across the four experimental groups: 18 in A, 18 in B, 17 in C, and 16 in D.

On top of that, 16 participants (out of the remaining 69)—14 BSc and 2 MSc students—mistakenly stopped the Tako plugin we installed to monitor their *implementation time* and *time to answer*. Thus, we consider these participants in all analyses related to *completeness* (RQ_1) and *code ownership* (RQ_2), but not in the time-based analyses (which relate to 53 participants). These 53 are also balanced across the groups: 13 in A, 13 in B, 14 in C, and 13 in D.

III. Results Discussion

We discuss the achieved results by RQ.

A. RQ_1: Impact on Productivity

The left-hand side of Fig. 2 shows the boxplots depicting the distribution of task *completeness* achieved by participants with (WAI) and without (NAI) the support of AI. The green boxes report the results when considering all participants, while

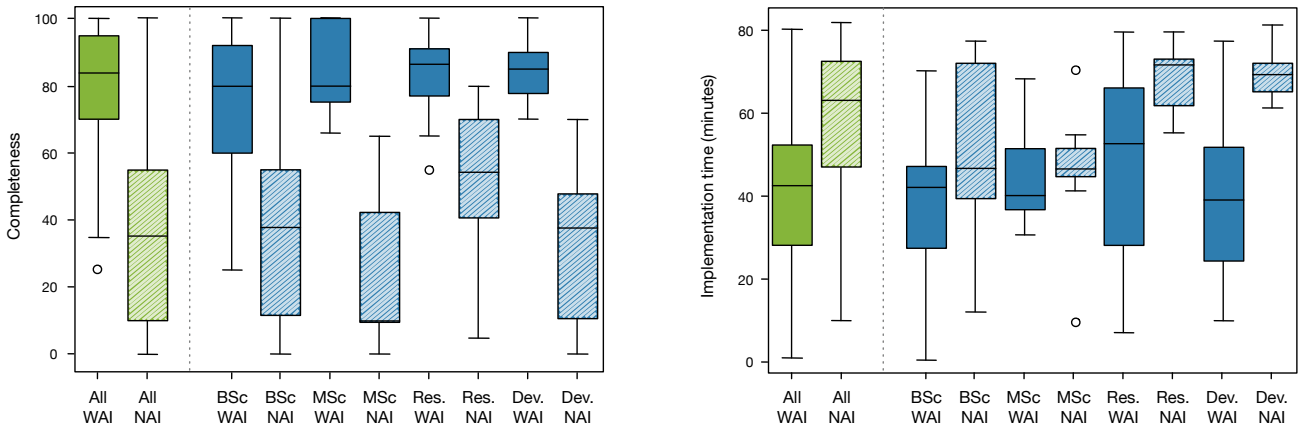

Fig. 2. RQ_1: Impact on Productivity.

the blue ones refer to four categories of participants (BSc, MSc, Researcher, Professional). To simplify the visualization, NAI distributions are characterized by patterned boxes. The right-hand side of Fig. 2 shows instead the distributions of *implementation time* (in minutes), following the same schema described for *completeness*.

The results in Fig. 2 show the strong boost in productivity provided to participants by the AI support. When considering all participants, the median *completeness* achieved without AI support is 35, compared to the 84 reached when AI was used. As per the *implementation time*, the median without AI is 63 minutes, as compared to the 43 minutes required with AI support. Such differences may look extreme, but can be partially explained by the tasks we designed for our experiment, which were non-trivial but, at the same time, could benefit from the AI support. Indeed, as explained in Section II, the tasks are related to quite popular domains (*e.g.,* games like Tic Tac Toe, content management systems relying on files as persistency mechanism) on which modern LLMs shine. Also, our findings are aligned with what reported in the literature: Weber *et al.* [24] report a +65% in requirements implemented in a given amount of time when developers use AI, while Peng *et al.* [25] document a 56% decrease in implementation time using Copilot.

It is interesting to see that the overall finding (*i.e.,* positive impact on productivity) is independent from the category of participants (*i.e.,* BSc, MSc, Researcher, Professional): All of them experienced a significant increase in *completeness* and a reduction in the *implementation time*. For some categories of participants the effect was more pronounced than for others, but this is not related to their level of experience. For example, professional developers benefited more from the AI support in terms of both *completeness* and *implementation time* as compared to BSc students (see Fig. 2). Despite appearing counterintuitive, this finding should be interpreted in light of the fact that each participant category was given tasks suited to their programming level, with researchers and professional developers dealing with the most complex tasks.

The linear mixed model in Table V shows the impact of

TABLE V
RQ_1 - COMPLETENESS: LINEAR MIXED-EFFECTS MODEL PREDICTING TASK COMPLETENESS. BOLD INDICATES $p < 0.05$. RANDOM-EFFECTS ENTRIES REPORT VARIANCE AND STD. DEV..

Random effects			
Groups name	**Var.**	**Std.Dev.**	
Participant	41.42	6.44	
Run	0.00	0.00	
Fixed effects			
Predictor	**Estimate**	**95% CI**	**p-value**
Treatment: WAI (vs NAI)	**42.99**	**[36.12, 49.86]**	**<0.001**
Education level: High school	5.68	[-12.68, 24.03]	0.538
Education level: Master	-6.40	[-22.15, 9.36]	0.420
Education level: PhD	-23.82	[-57.64, 10.00]	0.164
Position: Master	-17.89	[-53.05, 17.26]	0.312
Position: PhD Student	12.09	[-29.85, 54.02]	0.567
Position: Postdoc	35.72	[-21.66, 93.09]	0.218
Position: Professional dev.	4.04	[-35.47, 43.54]	0.839
Years of programming exp.	-0.03	[-2.06, 1.99]	0.974
Task: D1	-9.55	[-21.53, 2.43]	0.116
Task: D2	-4.32	[-38.58, 29.95]	0.802
Task: E2	-7.02	[-41.32, 27.28]	0.685
Task: D3	-24.44	[-60.26, 11.38]	0.178
Task: E3	-9.35	[-45.16, 26.47]	0.604
Task: D4	**22.50**	**[6.28, 38.72]**	**0.007**
Session: 2	**8.56**	**[1.78, 15.34]**	**0.014**

the random and fixed effects on the task completeness,[7] with the WAI treatment playing a significant role ($p < 0.001$). On average, WAI increased completeness by 42.99 points relative to NAI (95% CI [36.26, 49.72]), holding all covariates constant and accounting for clustering by participant and experiment.

As per the other independent (control) variables, two played a significant role in the achieved *completeness*, namely task D4 (p = 0.007, higher completeness as compared to that of the other tasks) and the second experimental session (p = 0.014, higher completeness as compared to the first session). Concerning task D4, it asked participants to write a test suite from scratch and this is the task for which we observed the strongest help given by AI: Participants using AI achieved

[7]Some of the tasks do not appear as control variable since being linearly dependent from other independent variables. Same for the language, which is captured by the "position" variable, since only Researchers and Professional developers worked with Python.

a median completeness of 100 (average 90), while those not using AI had a median of 42.5 (average 46). Basically, the highest completeness observed in D4 is influenced by the strong support given by the AI. As we will see, participants who took advantage of the AI in this task had a high price to pay in terms of code ownership (RQ_2).

Concerning the impact of the "session" variable, as explained in Section II-C1 we assess the robustness of the *task completeness* results to potential period and carryover effects through additional analyses. We introduced a Treatment × Session interaction to test whether the treatment effect differed across sessions. The interaction term was not statistically significant (p = 0.337), providing no evidence that the treatment effect depended on session order. Importantly, the main effect of Treatment remained significant in this model (estimate = 39.1, 95% CI [28.5, 49.6], $p < 0.001$). Also, we conducted a sensitivity analysis restricted to Session-1 observations only. In this conservative analysis, the treatment continued to have a significant effect on task completeness (estimate = 38.0, 95% CI [26.7, 49.2], $p < 0.001$). Taken together, these analyses indicate that the observed improvement in task completeness under AI assistance is robust to period effects and not attributable to carryover or session-order confounds.

Table VI shows the log-normal mixed effects model for the *implementation time*. The model yielded a boundary singular fit: The participant-level variance was estimated as 0.00 and the experiment-level variance as approximately 0.00. This indicates that, once accounted for all fixed predictors, little residual clustering remained at either level. Thus, although initially specified as a mixed-effects model to reflect the study design, for this outcome it effectively reduces to a model with negligible random-intercept contribution (*i.e.,* an ordinary regression model). To verify that inference was not an artifact of this specification, we fitted the corresponding ordinary regression model with the same fixed effects and obtained essentially identical estimates and p-values.

While the treatment was a significant predictor of implementation time (p = 0.003), things changed when introducing a Treatment × Session interaction to study a possible period effect. The interaction term was not statistically significant (p = 0.142), suggesting no strong evidence of differential treatment effects across sessions. However, the main effect of Treatment was no longer statistically significant (p = 0.263; estimate = -0.23, 95% CI [-0.64, 0.18]), indicating that the overall treatment difference was sensitive to the inclusion of period structure. Consistent with this finding, a Session-1-only sensitivity analysis, which provides an unbiased comparison free from carryover, also showed no significant effect of Treatment on implementation time (p = 0.719). These findings indicate that the apparent treatment effect on implementation time is not robust to period-based sensitivity analyses, and therefore we refrain from drawing strong conclusions regarding the impact of AI assistance on implementation speed.

Answer to RQ_1. AI support substantially boosts the productivity of participants, with a substantial boost in terms of task completeness in the given time. While there are tasks on which the AI helps more than others, the observed trends are valid across all tasks, languages, and participants' categories.

TABLE VI
RQ_1 - IMPLEMENTATION TIME: LOG-NORMAL MIXED-EFFECTS MODEL PREDICTING IMPLEMENTATION TIME. BOLD INDICATES P < 0.05. RANDOM-EFFECTS ENTRIES REPORT VARIANCE AND STD. DEV.

Random effects			
Groups name	**Var.**	**Std.Dev.**	
Participant	0.00	0.00	
Run	0.00	0.00	
Fixed effects			
Predictor	**Estimate**	**95% CI**	**p-value**
Treatment: WAI (vs NAI)	**-0.44**	**[-0.74, -0.15]**	**0.003**
Education level: High school	-0.52	[-1.16, 0.11]	0.106
Education level: Master	-0.06	[-0.54, 0.42]	0.796
Education level: PhD	-0.26	[-1.27, 0.76]	0.615
Position: Master	0.34	[-0.74, 1.41]	0.535
Position: PhD Student	0.28	[-2.28, 2.84]	0.826
Position: Postdoc	0.34	[-2.39, 3.08]	0.804
Position: Professional dev.	0.14	[-2.36, 2.65]	0.910
Years of programming exp.	-0.03	[-0.09, 0.04]	0.434
Task: D1	0.00	[-0.40, 0.40]	0.983
Task: D2	0.06	[-2.22, 2.35]	0.957
Task: E2	0.15	[-2.06, 2.36]	0.891
Task: D3	0.60	[-0.60, 1.79]	0.322
Task: E3	0.29	[-2.08, 2.66]	0.809
Task: D4	-0.30	[-2.05, 1.44]	0.732
Lines of Code	-0.00	[-0.00, 0.00]	0.633
Session: 2	0.00	[-0.25, 0.26]	0.991

B. RQ_2: Impact on Code Ownership

The left-hand side of Fig. 3 shows the boxplots depicting the distribution of percentage of correct answers (our proxy for *code ownership*) provided by participants after performing tasks with and without the support of AI. The right-hand side, instead, shows the time (in minutes) they spent answering the posed questions in the two treatments (WAI and NAI). The color/pattern schema used is the same described for the RQ_1's boxplots.

Visually, it can be seen that participants who implemented the code without AI support managed to correctly answer a higher percentage of technical questions. For example, when considering all participants (green boxplots), the median percentage of correct answers with the NAI treatment is 100%, as compared to the 87.5% they achieved with the WAI treatment. Such a trend is visible for all categories of participants, not only in terms of median, but with the whole distribution shifted towards higher values for the NAI treatment. There are, however, major differences observed across different types of participants, with some of them paying a higher cost than others in terms of *code ownership* when using the AI support. For instance, when looking at BSc students the difference in percentage of correct answers is minimal (median=87% in WAI and 88% in NAI, average=82% in WAI and 85% in NAI). One possible explanation for such a finding relates to the D3 and E3 tasks BSc students have been asked to perform: Both these tasks dealt with web APIs implemented with the Spring framework and offered less freedom in terms of implementation choices compared to the other tasks. For example, D3 required very specific API implementations for interacting with the GitLab server. As a result, these constraints left ○1

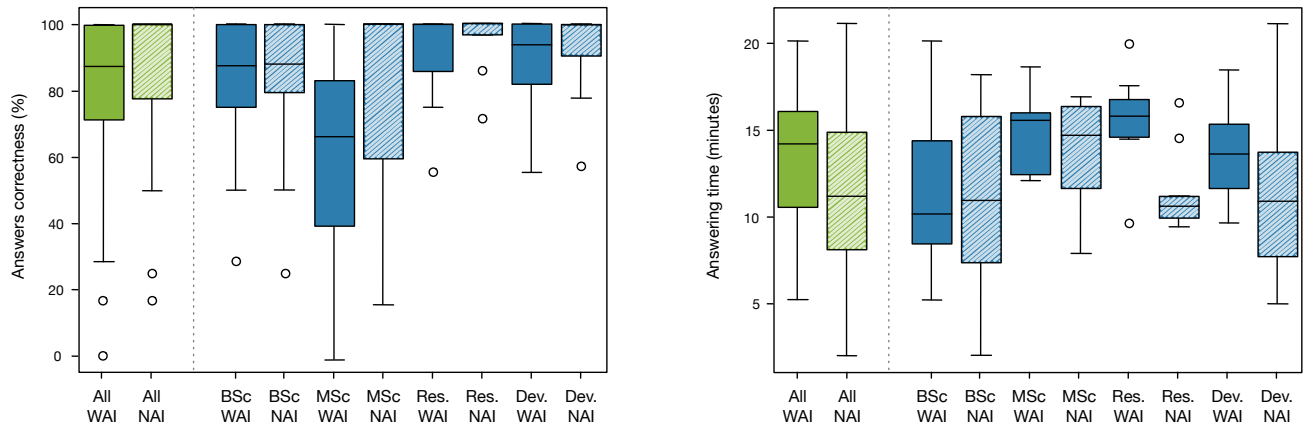

Fig. 3. RQ_2: Impact on Code Ownership.

with less "room" to generate complex questions about the implemented code and its design decisions, thus increasing the percentage of correct answers given by the BSc students.

When instead focusing on MSc students, they were the ones experiencing the strongest reduction in ownership as a result of the AI usage (median=67% in WAI and 100% in NAI, average=62% in WAI and 75% in NAI). Such a result can be traced back to our RQ_1 discussion, in which we observed the strong help AI provided to participants in task D4, which is indeed the one on which 12 out of the 13 MSc students involved in our study worked (all those from U2, while the additional MSc student is from U1). Indeed, if we only focus on this specific task, the impact on code ownership is even more dramatic: median=67% in WAI and 100% in NAI, average=55% in WAI and 93% in NAI. Thus, the major helping hand coming from the AI resulted in a severe loss of *code ownership*.

In the logistic mixed-effects model (Table VII), AI assistance was indeed associated with lower ownership correctness (OR = 0.58, 95% CI [0.38, 0.91], p = 0.016). In terms of absolute probabilities, the average marginal effect (*i.e.,* the change in probability of getting a correct answer when moving from the NAI to the WAI treatment) indicates a 5.95 percentage-point decrease in the probability of answering ownership questions correctly when using AI assistance (95% CI [-10.8, -1.1]). No other variable played a significant effect on the code ownership. Adding a Treatment $\times$ Session interaction provided no evidence of differential treatment effects across sessions (p = 0.473). In this model, the main effect of the treatment was attenuated and no longer significant, though the estimate remained similar (OR = 0.46, 95% CI [0.22, 1.00], p = 0.05). As per the Session-1-only sensitivity analysis, the treatment again showed a statistically significant effect on code ownership (OR = 0.39, 95% CI [0.16, 0.99], p = 0.047), consistent in direction and magnitude with the primary model. Overall, the negative impact of AI assistance on code ownership resulted to be robust to alternative specifications, with minor fluctuations in significance attributable to reduced statistical power rather than changes in effect direction.

We also performed a stratified analysis (open questions only *vs* closed questions only) to better assess whether the treatment effect on code ownership was mostly driven by specific types of questions. We accomplished this by replicating the logistic mixed-effects model on the subset of data related to each question type. The effect of treatment was significant in the model only considering open questions (p = 0.02), with OR = 0.45 (95% CI [0.23, 0.88]), but not for the one only considering closed questions (p = 0.09), with OR = 0.61 (95% CI [0.34, 1.08]). This is expected since statistical significance is not transitive under sample splitting, but the direction of the effect is the same for both strata.

TABLE VII
RQ_2 - CODE OWNERSHIP: LOGISTIC MIXED-EFFECTS MODEL (ODDS RATIOS) PREDICTING OWNERSHIP CORRECTNESS. BOLD INDICATES P $<$ 0.05. RANDOM-EFFECTS ENTRIES REPORT VARIANCE AND STD. DEV.

Random effects			
Groups name	**Var.**	**Std.Dev.**	
Participant	0.77	0.88	
Run	0.00	0.00	
Fixed effects			
Predictor	**OR**	**95% CI**	**p-value**
Treatment: WAI (vs NAI)	**0.58**	**[0.38, 0.91]**	**0.016**
Education level: High school	0.66	[0.18, 2.48]	0.539
Education level: Master	0.32	[0.08, 1.21]	0.092
Education level: PhD	0.26	[0.02, 3.60]	0.315
Position: Master	0.83	[0.06, 11.39]	0.889
Position: PhD Student	43.37	[0.31, 6071.38]	0.135
Position: Postdoc	116.65	[0.32, 42236.66]	0.113
Position: Professional dev.	48.22	[0.40, 5868.31]	0.114
Years of programming exp.	1.16	[0.97, 1.39]	0.100
Lines of Code	1.00	[1.00, 1.01]	0.529
Task: D1	0.47	[0.20, 1.09]	0.080
Task: D2	6.17	[0.08, 454.12]	0.406
Task: E2	8.83	[0.13, 611.88]	0.314
Task: D3	9.23	[0.62, 136.84]	0.106
Task: E3	16.12	[0.19, 1352.37]	0.219
Task: D4	4.05	[0.22, 73.41]	0.344
Question type: Open	1.00	[0.64, 1.56]	0.989
Session: 2	0.91	[0.63, 1.32]	0.628

A loss of code ownership when using the support of AI seems to also be visible in terms of *time needed to answer the questions*, at least when looking at the right-hand side of Fig. 3. Indeed, overall (green boxplots) we observe a higher answering time in WAI, with a +3 minutes in terms of median.

While this difference might seem modest, it translates to a ∼27% relative increase in answering time (median=14 minutes in WAI *vs.* 11 minutes in NAI). Also, this finding should be considered alongside the earlier results: After completing a task with AI support, participants spent more time answering questions compared to the NAI condition, yet achieved a lower percentage of correct responses. As per the findings by category of participants, BSc students were the only ones for which we did not observe an increase in the response time, likely for the characteristics of tasks D3 and E3 previously explained. For Researchers and Professional developers, the increase in answering time was instead quite noticeable. By leveraging the monitoring data collected through Tako [21], we isolated the set of `didChangeTextEditorUI` events performed in the IDE by participants while they were answering the questions. Such an event indicates a change of focus between the text editor views in the IDE which, during questions' answering time, were usually (i) the file containing the questions (and in which answers were written), and (ii) the implemented code files, which participants inspected when needed to answer the questions. On average, there were 87 of those events at the end of each task in the WAI treatment (median=44) and 34 in the NAI treatment (median=26). Since the number of switches between the text editor views may also be associated to the size of implemented program, usually larger for the WAI treatment, we also normalized the number of `didChangeTextEditorUI` events by LOC, obtaining an average of 0.55 events per line of code in the WAI treatment (median=0.25) versus a 0.37 in the NAI treatment (median=0.17).[8] This data suggests that *participants had to go back to the code more often when they used the AI support*, possibly indicating a loss of code ownership.

The log-normal mixed effects model (Table VIII) shows the impact of treatment on the answering time. The participant-level random-effect variance was estimated as 0.0517, while the experiment-level variance was small (0.0018), indicating modest residual clustering by participant and only minimal clustering by experiment after accounting for the fixed effects. The impact of treatment was significant (p = 0.010, estimate = 0.18, 95% CI [0.04, 0.31]). Also, introducing a Treatment × Session interaction did not reveal a significant interaction (p = 0.124), suggesting no strong evidence that the treatment effect varied by session. However, in this specification, the main effect of the treatment was no longer statistically significant (p = 0.788; estimate = 0.03, 95% CI [-0.20, 0.26]), indicating that the treatment difference was sensitive to the inclusion of session-specific effects. Similarly, a Session-1-only sensitivity analysis also showed no significant effect of the treatment on answering time (p = 0.441). Thus, the apparent effect of AI assistance on answering time is not robust to conservative period-based analyses, not allowing to draw strong conclusions regarding the impact of AI assistance on the time spent answering ownership questions.

Answer to RQ$_2$. The productivity benefits of AI-based tools in programming (RQ$_1$) are associated with reduced code ownership, as reflected in lower correctness on ownership questions, especially when AI support is highly effective. The evidence for an effect on answering time is less robust.

[8]Boxplots showing raw and normalized distributions of these events for both treatments are available in our replication package [11].

TABLE VIII
RQ$_2$ - ANSWERING TIME: LOG-NORMAL MIXED-EFFECTS MODEL PREDICTING QUESTION ANSWERING TIME. BOLD INDICATES $p < 0.05$. RANDOM-EFFECTS ENTRIES REPORT VARIANCE AND STD. DEV.

Random effects			
Groups name	**Var.**	**Std.Dev.**	
Participant	0.05	0.23	
Run	0.00	0.04	
Fixed effects			
Predictor	**Estimate**	**95% CI**	**p-value**
Treatment: WAI (vs NAI)	**0.18**	**[0.04, 0.31]**	**0.010**
Education level: High school	-0.10	[-0.52, 0.32]	0.638
Education level: Master	0.10	[-0.21, 0.42]	0.510
Education level: PhD	0.08	[-0.59, 0.76]	0.807
Position: Master	0.04	[-0.67, 0.74]	0.915
Position: PhD Student	0.11	[-1.30, 1.51]	0.882
Position: Postdoc	0.21	[-1.36, 1.78]	0.790
Position: Professional dev.	0.04	[-1.33, 1.41]	0.951
Years of programming exp.	-0.02	[-0.07, 0.03]	0.306
Task: D1	0.06	[-0.11, 0.24]	0.467
Task: D2	0.02	[-1.22, 1.27]	0.971
Task: E2	-0.03	[-1.25, 1.18]	0.955
Task: D3	-0.45	[-1.24, 0.34]	0.249
Task: E3	-0.12	[-1.42, 1.18]	0.853
Task: D4	0.50	[-0.38, 1.39]	0.260
Lines of Code	0.00	[-0.00, 0.00]	0.189
Session: 2	**-0.18**	**[-0.29, -0.07]**	**0.002**

IV. THREATS TO VALIDITY

Construct validity. The results of both RQs are partially based on data manually extracted from the code written by participants (task *completeness*) and the answers they provided (*code ownership*), thus involving subjectivity risks. To partially address them, we made sure that all manually-extracted data were checked by two authors. Also, authors were not aware of the specific treatment linked to the code and answers they inspected to reduce any form of experimenter bias.

Another threat relates to possible ceiling effects in the context of RQ$_2$. Code ownership under the NAI treatment frequently approached 100%, which may limit sensitivity to detect small differences between conditions. Despite this, we observed a statistically significant average marginal effect of the treatment (WAI *vs* NAI), -0.0595 (95% CI [-0.108, -0.011]), with a CI that excludes the 0. Possible ceiling effects would bias the AME toward 0. Thus, the observed -5.95% can be seen as conservative rather than inflated.

When fitting log-normal mixed-effects model for time-based measurements, we verified that log-transformation yielded approximately normal residuals and that model assumptions were reasonably satisfied by building Q-Q (quantile-quantile) plots. The latter compare the quantiles of model residuals to the quantiles of a normal reference distribution. Both the plots for *implementation* and *answering time* are available in our replication package [11] and show that residuals are approximately normal, with minor deviations at the lower and upper tail, supporting models adequacy.

In the WAI treatment, all participants were provided with GitHub Copilot enabled in the IDE. However, we did not collect detailed logs of AI tool usage, and participants were free to use additional AI-based tools as part of their normal workflow. As a result, we cannot precisely quantify the extent or nature of AI assistance used by individual participants. This limits our ability to attribute observed effects to specific tools or usage patterns within the AI-assisted condition and it is part of our future research agenda.

Finally, in our time-based analyses we had to exclude some participants due to missing time logs. To assess whether this could have biased our results, we compared participants included versus excluded from the time-based analyses using a logistic regression on observable covariates (*i.e.,* treatment, education level, position, years of experience in programming, task, session). The inclusion model indicated that only the "years of programming experience" was a significant predictor of having valid time logs: Each additional year of experience increased the odds of having valid time logs (OR=1.83). Importantly, treatment assignment was not associated with inclusion, suggesting that missingness is unlikely to bias the estimated treatment effects for time outcomes.

Internal validity. We considered several control variables in our analysis, to better isolate the impact of the studied treatment. Still, it is possible that other variables we did not control or analyze had an impact on our findings. Also, we experienced some form of "experimental mortality," with the need for excluding some participants from our study due to the reasons documented in Section II-D. This is not expected to affect our findings in any case since (i) we opted for a *within-subject* design, in which every participant performed an implementation task for both treatments, thereby always guaranteeing balanced participation across conditions; and (ii) as shown in Section II-D, the number of participants per experimental group (A, B, C, D) was still quite balanced.

A possible threat to internal validity is that the questions generated by the `o1` model for one treatment (WAI *vs.* NAI) were more difficult than those generated for the other treatment, thus affecting the results of RQ_2. To neutralize this threat, two authors independently rated a sample of 100 questions (50 from each treatment, 30 closed and 20 open questions) from 1 (simple) to 3 (difficult), without knowing the treatment they belong to. This resulted in an average score of 1.34 for WAI and 1.30 for NAI (author #1) and 1.50 for WAI and 1.48 for NAI (author #2), thus showing no significant difference in the difficulty of the questions across treatments. Nevertheless, we acknowledge that individualized question generation, while practically advantageous for adapting the assessment to each participant's implementation, inherently reduces the strict comparability of ownership scores across participants and conditions.

Finally, while all participants reported prior use of AI-based tools, we did not collect more detailed information about its extent or nature. This limits our ability to assess whether it influenced the observed effects.

External validity. We involved in our study 69 participants. Rather than reporting post-hoc power, we assess estimation precision under the final model specifications. For each primary outcome, namely *completeness* for RQ_1 and *code ownership* for RQ_2, we report standard errors and corresponding minimum detectable effects (MDEs), defined as $\pm 1.96 \times$SE, which indicate the smallest effects that could be reliably distinguished from zero. For *completeness*, MDE = $1.96 \times 3.436 = \pm 6.7$ points, which is reliable for the large differences we observed (AME=42.99, 95% CI [36.26, 49.72], indicating that moving from NAI to WAI increases expected task completeness by about 43%, on average. As per *code ownership*, MDE = $1.96 \times 0.025 = \pm 4.9$ points, being again reliable for the differences we observed (AME=-0.0595, 95% CI [-0.108, -0.011], indicating that WAI participants were estimated to be 5.95 percentage points less likely to answer correctly than NAI participants). Even so, we acknowledge the need for further replications, especially considering the differences in the magnitude of the results observed across different programming tasks.

Participant's position (*e.g.,* professional developers, bachelor student) was partially entangled with task characteristics and programming language, reflecting the practical constraints of the study design. As a result, we cannot fully disentangle experience effects from task-family or language-specific effects. While we include task indicators and conduct position-specific analyses, generalization beyond the studied cohorts, tasks, and languages should be made with caution.

We also performed an external-validity sensitivity analysis by re-estimating the models after excluding bachelor and master students and retaining only professionals and researchers. This restriction reduced the sample to approximately 35% of the original observations, resulting in a non-significant treatment effects when looking at the impact of treatment on code ownership. However, the estimated effect remained directionally consistent with the full-sample results with a wider confidence interval. In particular, for the code ownership analysis we got an OR=0.55, 95% CI [0.17, 1.79] for the professional&researchers-model, as compared to the OR=0.58, 95% CI [0.38, 0.91] for the full-sample model. Instead, the treatment effect remained significant ($p < 0.001$) for the completeness analysis.

Finally, considering the fast pace at which AI-based assistants for software developers are evolving, our findings may not generalize to newer agent-based tools recently proposed in the literature (see *e.g.,* [26]).

V. Related Work

Productivity. Several studies highlight the positive impact of AI assistants on developers' productivity. Bird *et al.* [27] conducted both a large-scale survey ($\sim$2k responses) and a small-scale case study (5 developers) on such a topic. Their overall results suggest that there is a significant reduction in task completion time when developers use GitHub Copilot.

Peng *et al.* [25] conducted an empirical study involving 95 professional programmers, finding that participants using Copilot completed coding tasks 55.8% faster than those without. Their findings also suggested greater benefits for less experienced developers, older programmers, and those who spend more hours coding per day, indicating AI tools' potential to facilitate career transitions into software development [25].

Weber *et al.* [24] conducted a user study with 24 developers aimed at comparing AI-assisted programming with non-AI-assisted programming. Their results highlight that productivity significantly increases with AI-based tools.

Ziegler *et al.* [28] highlighted the critical role of developers' perceived productivity in relation to their acceptance of Copilot suggestions, providing insights into how subjective productivity perceptions correlate with AI tool acceptance.

Sarkar *et al.* [29] explored the developer experience when using AI-driven code completion tools. Through interviews and usage analysis, the authors examine how AI assistance affects programming workflows, particularly focusing on developers' perception of productivity, cognitive load, and satisfaction. The findings reveal that while AI tools can speed up coding and support exploration, they also introduce cognitive overload and occasional interruptions, with developers expressing concerns about trust and over-reliance.

Compared to the aforementioned studies, our work sheds light on the *actual* productivity boost obtained by developers when using AI-based tools in a controlled experiment, rather than on their *perceived* productivity. We do so by measuring two proxies for productivity, namely the completeness of the coding task and the time taken to complete it. Furthermore, our study involves multiple categories of participants (BSc and MSc students, researchers, and professional developers), highlighting the differences observed among them. For example, while students observe a major boost in task completeness when using AI-based tools, they do not observe a significant reduction in the time taken to complete the task, differently from professional developers, who experimented the largest time savings. Besides studying productivity, we also analyzed one of the possible drawbacks of such a boost (*i.e.,* loss of code ownership).

Code Ownership. The impact of AI-based coding tools on code ownership is underexplored in the literature. We are aware of only one previous work on such an aspect: Weisz *et al.* [30] ran a survey involving 669 developers to understand (among other things) what is the *perceived* ownership of AI-generated code. Their results revealed that the majority of developers identify the AI-based tool as the sole author of the code when they do not modify its output.

Several studies investigated the impact of AI support on students and novice developers [31], [32], [33], [34], [35], with some reporting neutral or positive changes in students' learning outcomes [32], [33], and others reporting negative effects [31], [34], [35].

Related to our study but in a different domain is the recent work by Kosmyna *et al.* [7], which explored the neural and behavioral consequences (also in terms of ownership) of LLM-assisted essay writing (natural language). Participants were divided into three groups, allowed to use: (i) LLMs, (ii) search engines, and (iii) no tool at all (*Brain-only*). The self-reported ownership of essays was the lowest in the LLM group and the highest in the *Brain-only* group. Participants using LLMs showed less ability to quote their own essays just minutes after writing them. These findings align with our results, overall suggesting that AI-based tools may diminish users' sense of ownership and ability to explain or recall details.

Finally, recent work by Borg *et al.* [36] examines whether code developed with AI assistants impacts subsequent maintenance activities performed by developers who did not originally write the code. Through a controlled experiment run with 151 participants, they found no consistent signs of degraded code maintainability. However, the authors warn about the building of *cognitive debt* when relying on AI for code implementation: mental effort deferred in the short term may accumulate as long-term costs (*e.g.,* decreased creativity). Such an aspect, while complementary to the one of code ownership we focus on, also points to the need for carefully assessing the benefits of AI assistance during code development, not limiting it to the positive impact on productivity.

To the best of our knowledge, we are the first assessing the effects of AI-based tools on *actual* code ownership.

VI. Conclusion and Future Work

We ran a family of experiments involving 69 participants (34 BSc and 13 MSc students, 8 researchers, and 14 developers) aimed at assessing the impact of AI-based coding assistants on developers' productivity and code ownership. While AI tools significantly boost productivity, with higher task completeness and reduced implementation time, these benefits come with a measurable loss in code ownership, as shown by the lower percentage of correct answers despite the longer time spent answering questions. In fact, tasks on which AI assistance was most effective were also associated with the greatest reduction in code ownership. This serves as a warning as AI coding assistants keep getting better and better. The observed reduction in code ownership appears directionally consistent also in the professionals/researchers-only subsample, but the evidence there is substantially less precise. Thus, stronger confirmation in professionally focused samples is needed before drawing firm conclusions about real-world developer populations.

Our findings have important implications. First, in a real-world setting, reduced code ownership due to the use of AI assistants may hinder key code-related practices. For example, during *code review*, the contributor may be unable to argue for their implementation choices; while *debugging*, developers may struggle more in locating the cause of a failed test; and in *maintenance & evolution* activities, developers may require more time to implement change requests. Basically, the time saved during code implementation may be offset by increased effort in later stages of the software lifecycle, with consequences for team collaboration and project efficiency. However, the extent to which such downstream costs will materialize depends on how software development practices will adapt to AI assistance. It is plausible that future software workflows will increasingly delegate debugging and maintenance activities to AI agents, thereby reducing the need for developers to engage deeply with low-level code details. Such a "higher-level code understanding" is not fully captured by the comprehension-based ownership measures we used and, under such scenarios, reduced low-level code comprehension may not translate directly into reduced effectiveness. Nevertheless, even in this future, we expect human oversight, accountability, and collaboration remain central to software

engineering practice. Second, in educational contexts, while AI tools can lower entry barriers to programming, their use should be carefully moderated. Over-reliance on AI-generated code may hinder the development of deeper code understanding and problem-solving skills.

Finally, our study highlights a loss of code ownership in the very short-term (*i.e.,* right after the code has been implemented) when AI is used. Future studies should investigate long-term effects. For instance, when developers may need to re-familiarize themselves with their past code, *is past code easier to comprehend if it was written without AI support?* Furthermore, future work should also explore ways to improve the explainability of generated code and foster developer engagement with the suggestions. This may include mechanisms for encouraging active review of AI-generated output, thus minimizing a possible loss in code ownership.

ACKNOWLEDGMENTS

This article is part of the project PID2024-156482NB-I00, funded by MICIU/AEI/10.13039/501100011033 and by the ESF+. Tufano and Bavota acknowledge the financial support of the Swiss National Science Foundation for the PARSED project (SNF Project No. 219294).

REFERENCES

[1] "ChatGPT," https://chat.openai.com/, accessed: 2024-02-27.
[2] "Copilot website," https://copilot.github.com, accessed: 2022-11-10.
[3] R. Tufano, A. Mastropaolo, F. Pepe, O. Dabic, M. Di Penta, and G. Bavota, "Unveiling chatgpt's usage in open source projects: A mining-based study," in *Proceedings of 2024 IEEE/ACM 21st International Conference on Mining Software Repositories (MSR)*. IEEE, 2024, p. To Appear.
[4] S. Peng, E. Kalliamvakou, P. Cihon, and M. Demirer, "The impact of ai on developer productivity: Evidence from github copilot," 2023. [Online]. Available: https://arxiv.org/abs/2302.06590
[5] M. Greiler, K. Herzig, and J. Czerwonka, "Code ownership and software quality: A replication study," in *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, 2015, pp. 2–12.
[6] P. Thongtanunam, S. McIntosh, A. E. Hassan, and H. Iida, "Revisiting code ownership and its relationship with software quality in the scope of modern code review," in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 1039–1050. [Online]. Available: https://doi.org/10.1145/2884781.2884852
[7] N. Kosmyna, E. Hauptmann, Y. T. Yuan, J. Situ, X.-H. Liao, A. V. Beresnitzky, I. Braunstein, and P. Maes, "Your brain on chatgpt: Accumulation of cognitive debt when using an ai assistant for essay writing task," 2025. [Online]. Available: https://arxiv.org/abs/2506.08872
[8] Y. Yang, X. Xia, D. Lo, and J. Grundy, "A survey on deep learning for software engineering," *ACM Computing Surveys (CSUR)*, vol. 54, no. 10s, pp. 1–73, 2022.
[9] X. Hou, Y. Zhao, Y. Liu, Z. Yang, K. Wang, L. Li, X. Luo, D. Lo, J. Grundy, and H. Wang, "Large language models for software engineering: A systematic literature review," *ACM Transactions on Software Engineering and Methodology*, vol. 33, no. 8, pp. 1–79, 2024.
[10] "Openai o1," https://openai.com/o1/, 2024.
[11] "Replication package," https://github.com/seart-group/productivity-ownership-study, [n.d.].
[12] P. Sedgwick, "Convenience sampling," *BMJ*, 2013.
[13] S. J. Stratton, "Population research: convenience sampling strategies," *Prehospital and disaster Medicine*, vol. 36, no. 4, pp. 373–374, 2021.
[14] "Gitlab apis," https://docs.gitlab.com/api/rest/, [n.d.].
[15] "Building an application with spring boot," https://spring.io/guides/gs/spring-boot, [n.d.].
[16] L. Richardson and S. Ruby, *Restful web services*, 1st ed. O'Reilly, 2007.
[17] "Junit 5," https://junit.org/, [n.d.].
[18] "Remote Development - Visual Studio Marketplace," https://marketplace.visualstudio.com/items?itemName=ms-vscode-remote.vscode-remote-extensionpack, accessed: 2024-02-28.
[19] "Extension Pack for Java - Visual Studio Marketplace," https://marketplace.visualstudio.com/items?itemName=vscjava.vscode-java-pack, accessed: 2024-02-28.
[20] "Python - Visual Studio Marketplace," https://marketplace.visualstudio.com/items?itemName=ms-python.python, accessed: 2024-02-28.
[21] "Tako - Visual Studio Marketplace," https://marketplace.visualstudio.com/items?itemName=codelounge.tako, accessed: 2024-02-28.
[22] S. Schulhoff, M. Ilie, N. Balepur, K. Kahadze, A. Liu, C. Si, Y. Li, A. Gupta *et al.*, "The prompt report: A systematic survey of prompting techniques," 2024. [Online]. Available: https://doi.org/10.48550/arXiv.2406.06608
[23] R. Bartlett and F. Partnoy, "The ratio problem," *Available at SSRN: https://ssrn.com/abstract=3605606*, 2020.
[24] T. Weber, M. Brandmaier, A. Schmidt, and S. Mayer, "Significant productivity gains through programming with large language models," *Proceedings of the ACM on Human-Computer Interaction*, vol. 8, no. EICS, pp. 1–29, 2024.
[25] S. Peng, E. Kalliamvakou, P. Cihon, and M. Demirer, "The impact of ai on developer productivity: Evidence from github copilot," *arXiv preprint arXiv:2302.06590*, 2023.
[26] I. Bouzenia, P. Devanbu, and M. Pradel, "Repairagent: An autonomous, llm-based agent for program repair," in *Proceedings of the IEEE/ACM 47th International Conference on Software Engineering*, ser. ICSE '25. IEEE Press, 2025, p. 2188–2200. [Online]. Available: https://doi.org/10.1109/ICSE55347.2025.00157
[27] C. Bird, D. Ford, T. Zimmermann, N. Forsgren, E. Kalliamvakou, T. Lowdermilk, and I. Gazit, "Taking flight with copilot: Early insights and opportunities of ai-powered pair-programming tools," *Queue*, vol. 20, no. 6, pp. 35–57, 2022.
[28] A. Ziegler, E. Kalliamvakou, X. A. Li, A. Rice, D. Rifkin, S. Simister, G. Sittampalam, and E. Aftandilian, "Productivity assessment of neural code completion," in *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*, 2022, pp. 21–29.
[29] A. Sarkar, A. D. Gordon, C. Negreanu, C. Poelitz, S. S. Ragavan, and B. Zorn, "What is it like to program with artificial intelligence?" *arXiv preprint arXiv:2208.06213*, 2022.
[30] J. D. Weisz, S. V. Kumar, M. Muller, K.-E. Browne, A. Goldberg, K. E. Heintze, and S. Bajpai, "Examining the use and impact of an ai code assistant on developer productivity and experience in the enterprise," in *Proceedings of the Extended Abstracts of the CHI Conference on Human Factors in Computing Systems*, 2025, pp. 1–13.
[31] G. Jošt, V. Taneski, and S. Karakatič, "The impact of large language models on programming education and student learning outcomes," *Applied Sciences*, vol. 14, no. 10, p. 4115, 2024.
[32] Y. Xue, H. Chen, G. R. Bai, R. Tairas, and Y. Huang, "Does ChatGPT help with introductory programming? An experiment of students using ChatGPT in CS1," in *Proceedings of the 46th International conference on software engineering: software engineering education and training*, 2024, pp. 331–341.
[33] M. Kazemitabaar, J. Chow, C. K. T. Ma, B. J. Ericson, D. Weintrop, and T. Grossman, "Studying the effect of AI code generators on supporting novice learners in introductory programming," in *Proceedings of the 2023 CHI conference on human factors in computing systems*, 2023, pp. 1–23.
[34] J. Prather, B. N. Reeves, J. Leinonen, S. MacNeil, A. S. Randrianasolo, B. A. Becker, B. Kimmel, J. Wright, and B. Briggs, "The widening gap: The benefits and harms of generative ai for novice programmers," in *Proceedings of the 2024 ACM Conference on International Computing Education Research-Volume 1*, 2024, pp. 469–486.
[35] P. Vaithilingam, T. Zhang, and E. L. Glassman, "Expectation vs. experience: Evaluating the usability of code generation tools powered by large language models," in *CHI Conference on Human Factors in Computing Systems Extended Abstracts*, 2022, pp. 1–7.
[36] M. Borg, D. Hewett, N. Hagatulah, N. Couderc, E. Söderberg, D. Graham, U. Kini, and D. Farley, "Echoes of ai: Investigating the downstream effects of ai assistants on software maintainability," 2025. [Online]. Available: https://arxiv.org/abs/2507.00788