Echoes of AI: Investigating the Downstream Effects of AI Assistants on Software Maintainability

Markus Borg[1,3*], Dave Hewett[2], Nadim Hagatulah[3], Noric Couderc[3], Emma Söderberg[3], Donald Graham[4], Uttam Kini[5], Dave Farley[6]

[1*]CodeScene, Malmö, Sweden.
[2]Equal Experts, London, UK.
[3]Dept. of Computer Science, Lund University, Lund, Sweden.
[4]Equal Experts, Cape Town, South Africa.
[5]Equal Experts, Bengaluru, India.
[6]Continuous Delivery, London, UK.

*Corresponding author(s). E-mail(s): markus.borg@codescene.com;
Contributing authors: dave.hewett@equalexperts.com;
nadim.hagatulah@cs.lth.se; noric.couderc@cs.lth.se;
emma.soderberg@cs.lth.se; donald.graham@equalexperts.com;
uttam.kini@equalexperts.com; info@continuous-delivery.co.uk;

Abstract

[Context] AI assistants, like GitHub Copilot and Cursor, are transforming software engineering. While several studies highlight productivity improvements, their impact on maintainability requires further investigation. [Objective] This study investigates whether co-development with AI assistants affects software maintainability, specifically how easily other developers can evolve the resulting source code. [Method] We conducted a two-phase, preregistered controlled experiment involving 151 participants, 95% of whom were professional developers. In Phase 1, participants added a new feature to a Java web application, with or without AI assistance. In Phase 2, a randomized controlled trial, new participants evolved these solutions without AI assistance. [Results] Phase 2 revealed no significant differences in subsequent evolution with respect to completion time or code quality. Bayesian analysis suggests that any speed or quality improvements from AI use were at most small and highly uncertain. Observational results from Phase 1 corroborate prior research: using an AI assistant yielded a 30.7% median reduction in completion time, and habitual AI users showed an estimated 55.9% speedup. [Conclusions] Overall, we did not detect systematic maintainability advantages or disadvantages when other developers evolved code co-developed with AI assistants. Within the scope of our tasks and measures, we observed no consistent warning signs of degraded code-level maintainability. Future work should examine risks such as code bloat from excessive code generation and cognitive debt as developers offload more mental effort to assistants.

Keywords: software engineering, programming with AI, controlled experiment, maintainability, code quality

1 Introduction

Generative AI is rapidly transforming software development, disrupting the discipline as we know it. Tools based on Large Language Models (LLMs), such as GitHub Copilot and ChatGPT, have seen widespread adoption among developers (JetBrains, 2024). The former exemplifies an IDE-integrated code completion assistant, while ChatGPT represents a general-purpose tool that supports chat-based programming. The appeal of AI assistants for code synthesis is clear and, as we will review in Section 2.3, several empirical studies, in fact, suggest that working with them can lead to significant productivity gains.

For many developers, AI assistants are now a natural part of the development context. The first secondary studies on their use have now been published, e.g., reviews by Ani et al (2024) and Husein et al (2025), which summarize the risks of LLM-based code synthesis identified in primary studies. Risks include the generation of incorrect code, the introduction of security vulnerabilities, unnecessarily complex code completions, and reduced code maintainability. In this study, we focus on the last aspect: how AI-assisted development influences maintenance and evolution.

As many organizations move into a hybrid world where human- and machine-generated code coexist, we urgently need to understand how the use of AI assistants affects maintainability. Recent public announcements from Microsoft, Meta, and Google state that roughly a third of their new code is already AI-generated (Prosser, 2025), demonstrating that the change is happening now. Thus, we argue that maintainability has never been more important since the sheer volume of code will increase rapidly from this point on. Furthermore, we posit that in the foreseeable future, human developers must remain able to manually evolve source code no matter its provenance.

We conduct a preregistered two-phase controlled experiment (Borg et al, 2024b). Our starting point is that maintainable code should be easy to reason about and modify by someone other than the original author. In Phase 1, participants extend a Java web application with or without the help of an AI assistant. Phase 2 is a Randomized Control Trial (RCT), in which new participants are randomly assigned to evolve a solution from Phase 1 *without using AI assistants*. Accordingly, we primarily assess maintainability by the ease with which a new developer can add features to existing code. To complement this perspective, we include CodeScene's CodeHealth (Tornhill and Borg, 2022) and test coverage measurements to broaden the assessment. Finally, we measure perceived productivity guided by the SPACE framework (Forsgren et al, 2021) and collect rich free-text data through exit questionnaires.

For an experiment relying on volunteers from a wide range of organizations, our study is unique in both the number of participants and their level of seniority. Our Phase 2 RCT provides no clear evidence that code co-developed with AI assistants is more efficient to evolve manually, and any speed advantage in the manual evolution task was small and statistically unreliable. For code quality, we found no significant differences in CodeHealth between treatment and control, although Bayesian analysis suggests a small CodeHealth improvement when the original solution was co-developed with AI by a habitual AI user. Although not the primary focus of our study, the Phase 1 results corroborate previous findings: AI assistants significantly reduced task completion time, with a median improvement of 30.7%. Moreover, the posterior mean effect for habitual AI users among the participants was 55.9%.

Coding agents, introduced as the third-generation AI assistants in Section 2.2, are now on the rise, offering autonomous code synthesis based on a developer's intent. Our study was conducted in late 2024, so its empirical results predate this trend. Instead, our study might be one of the last to include a large sample of developers who had not yet been fundamentally affected by AI assistants. There is a clear need for additional empirical studies to investigate these agentic trends, both controlled experiments like ours and longitudinal case studies, to explore direct and indirect effects on maintainability.

The rest of the manuscript is organized as follows:

- Section 2 introduces fundamental constructs, presents different generations of AI assistants for code, and reviews related empirical studies.
- Section 3 describes the research method, including data collection and analysis.
- Section 4 presents the results from the RCT in Phase 2 and observational findings from Phase 1.
- Section 5 discusses the results in light of previous work and shares our interpretation of implications for practice.
- Section 6 discusses the limitations of our study and its threats to validity.
- Section 7 concludes the paper and outlines directions for future work.

2 Background and Related Work

This section first introduces background on software maintainability and productivity. Second, we present the disruptive trend of AI-assisted development. Finally, we review empirical studies on the impact of AI assistants.

2.1 Software Maintainability and Productivity

The two constructs maintainability and productivity are cornerstones of this research. As both are complex and multi-faceted, this section describes how we rely on state-of-the-art approaches to measure them.

The ISO 25010 quality model defines maintainability as *"the degree of effectiveness and efficiency with which a product or system can be modified to improve it, correct it or adapt it to changes in environment, and in requirements"* (International Organization for Standardization, 2011). To acknowledge the different constituents of this quality, ISO 25010 further defines five sub-qualities: 1) modularity, 2) reusability, 3) analyzability, 4) modifiability, and 5) testability. We use the top-level maintainability definition in this work, and argue that completion time on an adaptation task is a useful proxy for maintainability, since more maintainable code should require less effort and time to evolve.

A related concept is Technical Debt (TD), defined as: *"In software-intensive systems, TD is a collection of design or implementation constructs that are expedient in the short term but set up a technical context that can make future changes more costly or impossible. Technical Debt presents an actual or contingent liability whose impact is limited to internal system qualities, primarily maintainability and evolvability"* (Avgeriou et al, 2016) Note that this definition includes the term evolvability, which is not explicitly mentioned in ISO 25010. We rely on its definition by Cook et al (2003) as *"the capability of software products to be evolved to continue to serve its customer in a cost-effective way"* and consider it subsumed by the ISO 25010 maintainability definition.

The presence of code smells is widely recognized as a factor that degrades maintainability. Code smells, a term (like TD) coined by the agile community, refer to problematic code that developers should preferably refactor – typically to make it easier to understand and extend (Mantyla and Lassenius, 2006). The academic literature on code smells is extensive, to the point that even a tertiary study has been published (Lacerda et al, 2020). Thanks to extensive lists of code smells, and capable detection tools, maintainability can pragmatically be treated as a negative quality, i.e., the degree to which code does not contain smells.

CodeScene measures maintainability using the CodeHealth™ metric. The file-level score penalizes the presence of 25+ code smells[1] known to increase cognitive load on developers. Our previous work shows that CodeHealth correlates with increased maintenance costs and defect risks (Tornhill and Borg, 2022; Borg et al, 2024c), as well as slower onboarding (Borg et al, 2023). Furthermore, CodeHealth outperforms competing metrics (Borg et al, 2024a), including SonarQube's Maintainability Rating and Microsoft's Maintainability Index on the Maintainability Dataset – a benchmark developed by Schnappinger et al (2020).

CodeHealth is a numeric value between 1 and 10. The metric aligns with Fenton (1994)'s seminal work on software measurement, particularly the philosophy that the best way to assess code complexity is by identifying and quantifying specific complexity-inducing attributes. A file with no detected code smells receives a perfect score of 10. For each code smell that is detected, the file gets a decreased CodeHealth score – infinitesimally approaching 1 for the very worst files. When aggregating CodeHealth across multiple files, CodeScene reports a file-size-weighted average of

[1]Some are language- or paradigm-dependent, thus not an exact number.

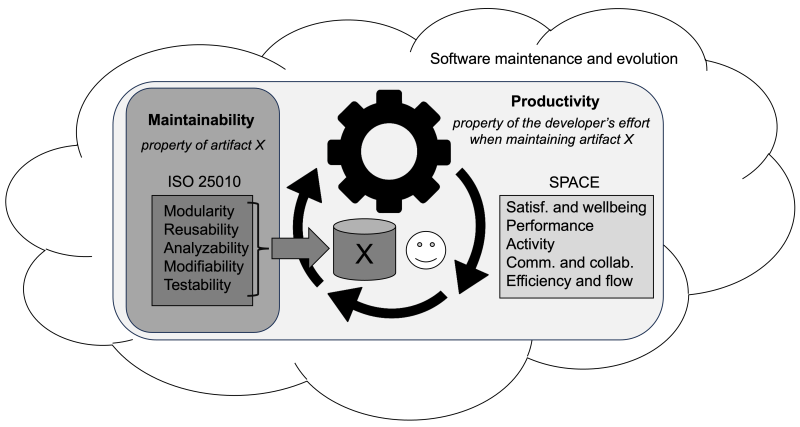


Fig. 1: Conceptual relationship between maintainability (artifact-centric) and productivity (developer-centric). ISO 25010 frames maintainability, SPACE frames productivity, and artifact X links the two through the maintenance activity.

the file-level values. In our study, we complement task completion time with CodeHealth to better measure the maintainability construct. Note that we customize the CodeHealth scoring for increased sensitivity in this study, as described in Section 3.2.

Developer productivity is notoriously difficult to define. Numerous publications have addressed the topic since the early days of software engineering (Sadowski and Zimmermann, 2019), and there is broad agreement that productivity is inherently multi-dimensional (Jaspan and Sadowski, 2019). Yet, many naïve myths and simplistic metrics continue to circulate in the software industry. In this study, we rely on the SPACE framework by Forsgren et al (2021) to discuss productivity in the context of software maintenance and evolution. Rather than providing an overarching definition of productivity, SPACE decomposes the construct into five dimensions:

- **S atisfaction and wellbeing.** How fulfilled developers feel with their work, team, tools, and culture. Also, how their work impacts their happiness and health.
- **P erformance.** The outcome of a process. Note that outcome goes beyond mere output, as in "Did the contributed code reliably do what it was supposed to do?"
- **A ctivity.** The volume of actions or outputs completed while performing work, such as commits or issues closed.
- **C ommunication and collaboration.** How developers exchange information and coordinate work, including perceptions of effectiveness in information seeking.
- **E fficiency and flow.** The ability to complete work or make progress on it with minimal interruptions or delays.

Figure 1 depicts how we position maintainability and productivity in this study. The cloud represents the overall context. In dark gray, we show maintainability as a property of artifact X, characterized by the ISO 25010 sub-characteristics, which we treat jointly in this work. In light gray, we position productivity as a reflection of a developer's effort when carrying out an adaptation task on the same artifact. The five SPACE dimensions frame our discussion on productivity in this context.

As evident from the SPACE research, the personal perception of productivity plays a central role, i.e., the three dimensions **S**, **C**, and **E** have large subjective components. In this study, we refer to this subset of productivity dimensions as *perceived productivity*. Following previous work on productivity with GitHub Copilot by Ziegler et al (2024), we measure this using Likert scales. These are used both to pre-screen participants' preference for working with AI assistants and to assess their experience with the development tasks.

2.2 AI-assisted Software Development

The development of AI assistants is evolving at an astonishing pace. We organize this section into three generations of AI assistants as outlined by Kim and Yegge (2025)

The **first generation** of AI assistants refers to LLM-driven *code completion* tools. The most well-known is undoubtedly GitHub Copilot[2], released in October 2021 – GitHub recently communicated (May 2025) that it now has 15 million users. Other tools in this category include TabNine[3], Google Gemini Code Assist[4], and IBM's watsonx Code Assistant[5]. These tools have in common that they are 1) largely reactive and 2) limited to local context. For example, code completion tools can provide next-token predictions or fill in method signatures based on nearby code. A grounded theory study by Barke et al (2023) found that the interaction is bimodal. In *acceleration mode*, developers know what to do and complete it faster with the assistant; in *exploration mode*, they are instead uncertain and use the assistant to quickly explore options. Academic evaluations of this generation primarily focused on Completion Acceptance Rates (Ziegler et al, 2022), which are far from the maintainability and productivity constructs central to our study.

The **second generation** of AI assistants focuses on enabling *chat-based programming*. The interaction follows the prompt-response mode popularized by OpenAI's ChatGPT[6], released in November 2022. Although launched as a general-purpose assistant, ChatGPT quickly demonstrated impressive coding capabilities. While many users still copy-paste from web browsers, specialized IDEs such as Cursor[7] and Windsurf[8] soon emerged to bring chat-based functionality into the developers' environment. This generation of tools is typically characterized by 1) responding to natural language queries and 2) operating with larger context windows than previous code completion tools. Compared to the first generation, chat-based assistants are often used to generate larger amounts of code, including full functions and boilerplate based on comments. Empirical studies on second-generation tools have typically focused on the generated code rather than developer interactions (Treude and Storey, 2025) – our study provides new insights to fill this gap.

The **third generation** of AI assistants refers to *autonomous coding agents*. This is a rapidly evolving space, with several vendors actively competing at the time of this

[2]https://github.com/features/copilot
[3]https://www.tabnine.com/
[4]https://codeassist.google/
[5]https://www.ibm.com/products/watsonx-code-assistant
[6]https://chatgpt.com/
[7]https://cursor.com/
[8]https://windsurf.com/

writing. Notable examples are Anthropic's Claude Code[9], Sonargraph's Amp[10], and OpenAI's Codex[11] (a name previously referring to an LLM trained on code). These agents accept high-level tasks from developers and follow plan–execute–validate–revise loops until a stopping condition is met. Compared to previous generations, these AI tools are distinguished by 1) persistent memory across chat interactions, 2) better understanding of project-level context, and 3) integration with other development tools such as linters, test runners, and documentation systems. Interoperability is often enabled via command-line interfaces or the Model Context Protocol (MCP) (Anthropic, 2024). As reported in Section 1, our study largely predates the third generation tools.

2.3 Empirical Studies on AI-Assisted Development

Our current study adds to a growing body of empirical research on AI-assisted development. In this section, we focus on studies where participants 1) complete realistic tasks, and 2) evaluations go beyond Completion Acceptance Rates and simplistic code metrics such as keystrokes and Lines of Code (LoC). We include both controlled experiments and field surveys that capture developers' real-world experiences.

We provide the first RCT that explicitly targets the maintainability of code written by AI-assisted developers. In contrast, the most closely related prior studies have focused on developer productivity, pioneered by GitHub's internal research on GitHub Copilot. The first and most widely cited work is the controlled experiment by Peng et al (2023), which recruited freelance programmers (N=95) to implement an HTTP server in JavaScript. Participants were randomly assigned to either a treatment group with GitHub Copilot or a control group without assistance. The authors report that the treatment group completed the task 55.8% faster on average. Although this study generated substantial attention and was used in marketing, it remains unpublished in a peer-reviewed venue.

A year later, another GitHub team published a paper combining repository mining of usage data with a questionnaire-based survey (N=2,047) (Ziegler et al, 2024). The results included usage telemetry and survey responses structured around the SPACE framework described in Section 2.1, which we adopt in our current study. By sending the survey to 17,240 users of the GitHub Copilot free technical preview, the authors captured experience from real usage in early 2022. Their findings suggest that the AI assistant had a significant positive impact on perceived productivity across multiple dimensions, e.g., task completion time, quality, cognitive load, enjoyment, and learning. Notably, the reported gains were higher among junior developers.

IBM Research conducted a similar questionnaire-based survey among users of the internal AI assistant watsonx Code Assistant (WCA) (Weisz et al, 2025). Instead of using the dimensions of SPACE, they designed the instrument to collect attitudinal measures of productivity and surveyed participants (N=669) from a WCA training program. The authors report that the main WCA use case was code comprehension rather than generation. While the respondents generally felt that their work was faster,

[9]https://www.claude.com/product/claude-code
[10]https://sourcegraph.com/amp
[11]https://openai.com/codex/

easier, and of higher quality with WCA, the magnitudes were small. Moreover, the individual differences were substantial – 42.6% felt that WCA made them less effective.

Google has also published results from related internal research activities. Paradis et al (2025) conducted an RCT (N=96) with Google developers to assess the impact of three AI features on the completion time of a realistic C++ task: 1) code completion, 2) smart paste, and 3) chat-based programming. Participants were randomly assigned to either the treatment group with all three features enabled, or the control group with all disabled. The authors report a 21% average speedup with AI support, although this effect was not significant when controlling for developer proficiency and task familiarity. In contrast to previous work, they found that more senior participants were faster with AI than juniors. The task under study in our RCT is larger in terms of both LoC and expected task completion time.

Chatterjee et al (2024) report on a six-week controlled experiment (N≈100[12]) at ANZ Bank, in which participants completed algorithmic programming tasks in Python. After two weeks of GitHub Copilot training, the study design included both between-group and within-subject comparisons. The results show that participants using the AI assistant completed tasks 42.3% faster on average, with developers less proficient with Python benefiting the most. Furthermore, the authors report that the code quality improved, with fewer bugs and code smells. Finally, participants shared that Copilot supported them in understanding, testing, and documenting code.

Weber et al (2024) conducted a controlled experiment (N=24) using a within-subjects design in which participants completed three Python programming tasks. Each task was paired with one level of AI-assistance: GitHub Copilot, chat-based programming (GPT-3), or traditional web search. The authors report that both types of AI-assistance led to more completed requirements per minute, greater code output, and higher self-reported satisfaction. Compared to our study, we note that the sample was small, the tasks were short, and only nine participants were professional developers, with an average age of 26.8 years, indicating a mostly junior cohort.

Liang et al (2024) conducted a questionnaire-based survey in January 2023 (N=410) with a diverse set of AI-assisted developers to study usability. GitHub Copilot was the most widely used tool (74.6%), whose users reported that a median of 30.5% of their code was now generated by the AI assistant. The primary motivations for adopting AI assistants were 1) reducing keystrokes, 2) completing tasks faster, and 3) skipping web searches for code examples. On the other hand, the most common reasons for not using such tools were 1) that the code output often is of subpar quality, and 2) the difficulty of controlling the AI assistants to produce the desired output. Our prescreening survey (N=449, see Section 4.1) is of similar size, but features a notably higher proportion of professional developers (92.2% vs. 49.5%).

Butler et al (2025) conducted a mixed-methods study at a multinational company to examine the impact of introducing GitHub Copilot to new users. The authors combined an RCT (N=51) with surveys, telemetry analysis, and a three-week diary study (N=106). A difference-in-difference analysis of telemetry data found no statistically significant differences, but medium to large effect sizes for most of the metrics. However, the surveys and diaries showed interesting subjective experiences: 84% reported

[12]Some methodological details are not fully disclosed in that paper.

positive changes in their daily work, e.g., reduced web searching and less manual boilerplate coding. Moreover, 66% described feeling more positive about work after introducing GitHub Copilot.

Finally, Cui et al (2025) conducted a massive field experiment combining three RCTs (N=4,867) at Microsoft, Accenture, and an anonymous Fortune 100 company. Participants were were randomly assigned to use GitHub Copilot or serve in the control group without access. During a two to eight month period, participants using the AI assistant completed 26% more tasks per week, with larger effects among junior developers. While the sample size is outstanding, no direct measurements of code quality measurements were collected – only build success rate as a simple proxy. It was significantly lower at Accenture with a medium effect size, but there was no difference in the pooled results.

In summary, several early studies were conducted by the major tech companies Microsoft, IBM, and Google, leveraging unique access to AI assistants. Across both experiments and survey research, studies consistently report productivity gains from AI assistants – typically on the order of 20–30% faster task completion for professionals, and sometimes more for novices or repetitive tasks. However, findings related to code quality and its potential impact on maintainability are less clear. Previous studies did not follow up with a second developer and rarely employed objective maintainability proxies. Our study is designed to address this gap by focusing on downstream evolution tasks and by grounding our analysis in established metrics and frameworks.

3 Method

Our goal is to investigate the impact of AI assistants on software maintainability. We break down the goal into two research questions: RQ1) Do developers manually evolve code that has been co-developed with AI assistants more efficiently? and RQ2) Does code co-developed with AI assistants result in higher quality upon manual evolution? We rely on four metrics to answer these RQs: task completion time, perceived productivity according to the SPACE framework, CodeHealth, and test coverage. Fig. 2 summarizes the aim of this study, structured using the GQM model (Basili et al, 1994). Our preregistered design was guided by the ACM SIGSOFT Empirical Standard for Experiments with Human Participants (ACM SIGSOFT, 2024b) and we used its essential attributes as a reporting checklist when writing this manuscript.

3.1 Study Design and Participant Recruitment

Figure 3 shows an overview of our preregistered, two-phase sequential design. In Phase 1, half of the participants prepare artifacts to be used by either the treatment group or the control group. In Phase 2, the remaining participants take part in an RCT, in which all participants receive only one treatment. The rest of this section is organized according to the Steps A] – E] indicated in the figure, and the experimental variables on the right-hand side are defined in Section 3.2.

Step A] We called for volunteers to take part in a controlled experiment about "Software Development with AI Assistants" (see A] in Fig. 3). All participants volunteered to engage in the research, and agreed to complete assigned tasks remotely in

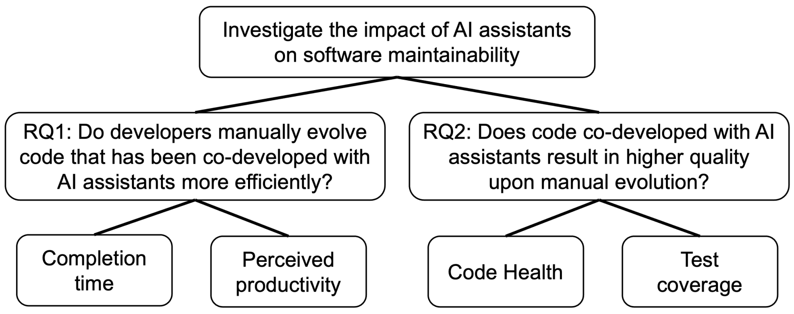

Investigate the impact of AI assistants on software maintainability
RQ1: Do developers manually evolve code that has been co-developed with AI assistants more efficiently?
RQ2: Does code co-developed with AI assistants result in higher quality upon manual evolution?
Completion time
Perceived productivity
Code Health
Test coverage

Fig. 2: Goal of the study, outlined using the GQM structure.

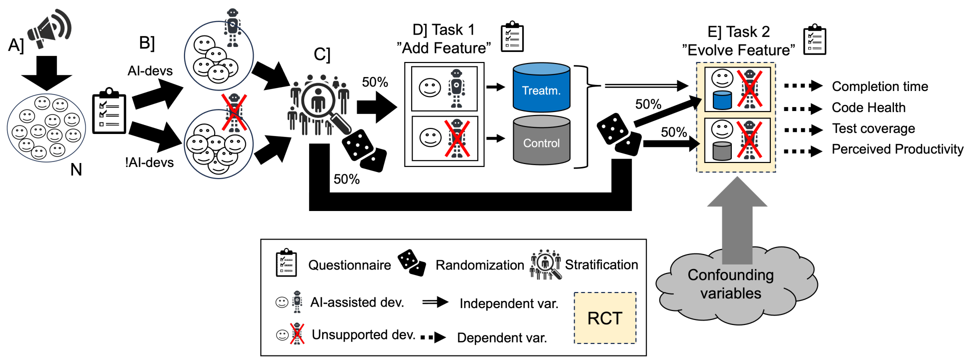


Fig. 3: Overview of the study. The part in the yellow box, to which about 50% of the participants were assigned, constituted the RCT.

the preferred development environment. We recruited participants through i) social media advertisements on platforms such as YouTube, LinkedIn, and X and ii) using our personal networks. To incentivize participation, all participants were offered a signed copy of the last author's latest book (Farley, 2022), and a chance to win a private online "Ask Me Anything" session with him.

Step B] Participants signed up for the study by completing the pre-screening questionnaire outlined in Table 1. In total, 449 participants submitted valid questionnaires. The primary purpose of the pre-screening questionnaire was to facilitate subsequent stratified random sampling into Phase 1 or Phase 2. In Phase 1, we aimed to assign an equal share of participants to either use AI assistants or work without any AI support. We refer to these cohorts as **AI-devs** and **!AI-devs**, respectively. In Phase 2, all participants were instructed to work without AI support.

To facilitate the assignment, the pre-screening collected participant information about 1) experience with AI assistants and 2) whether such assistance was currently the preferred way of working (measured using a Likert scale). To qualify for the **AI-devs** cohort, participants had to: i) answer yes to Q1-6, ii) agree to statement Q1-7a), and iii) have a positive mean response to the preference questions Q1-7 b)–i) – taking the inverted question g) into account.

Table 1: Outline of the pre-screening questionnaire. A colon indicates a free-text input field. A letter within parentheses shows a mapping to the corresponding SPACE dimension – item g) in Q1-8 is inverted. As will be explained in Section 3.2.2, we refer to participants who strongly agree to Q1-7a as "habitual AI users."

Question	**Type**	**Operationalization**
Q1-1. What is your gender?	Nominal	a) Man b) Woman c) Non-binary d) Prefer not to disclose, e) Prefer to self-describe:
Q1-2. What is your age?	Ordinal	a) 19 or younger b) 20-29, ... g) 70 or older, h) Prefer not to disclose
Q1-3. Where do you live?	Nominal	Closed country list + Prefer not to disclose
Q1-4. Which of the following best describes what you do?	Nominal	a) Student, full-time or part-time, b) Professional programmer, writing code for work, c) Hobbyist programmer, writing code for fun or outside of work, d) Researcher, e) Prefer not to disclose, f) Other:
Q1-5. How proficient are you in software development with Java?	Ordinal	a) Beginner, I can write a correct implementation for a simple function b) Intermediate, I can design and implement whole programs, c) Advanced, I can design and implement a complex system architecture
Q1-6. Do you have experience of working with an AI assistant while programming?	Binary	a) Yes, b) No (ends the questionnaire)
Q1-7. Experience and preferences	5-point Likert + N/A	Thinking of your experience as a developer and your ways of working, please indicate your level of agreement with the following statements. a) I am a habitual user of AI assistants while programming. b) I am more productive when using AI assistants. (E) c) I complete tasks faster when using AI assistants. (E) d) I spend less time searching for information or examples when using AI assistants. (C) e) I complete repetitive programming tasks faster when using AI assistants. (E) f) Using AI assistants helps me stay in the flow. (E) g) Using AI assistants is distracting. (E) h) I feel more fulfilled with my job when using AI assistants. (S) i) I can focus on more satisfying work when using AI assistants. (S)

Allowing participants to adhere to their preferred work methods reduces the likelihood of protocol non-compliance, such as using AI tools when told not to. However, this flexibility might introduce a bias, e.g., less experienced developers could be more inclined to use AI assistants. To compensate for this, we will include Java proficiency as a covariate in our Bayesian model. Note that early research suggests that junior developers might benefit more from AI assistance (Ziegler et al, 2024).

A total of 118 participants (26.2%) who answered confirmatory to both 1) and 2) were split into the **AI-devs** cohort (see B] in Fig. 3 and Section 3.3.1 for details). The remaining 331 participants were assigned to the **!AI-devs** cohort. Notably, the proportion of **AI-devs** (26.2%) was closely aligned with our ideal target of one quarter of the participants in this cohort (Borg et al, 2024b). As AI-assisted development becomes increasingly prevalent, finding developers without prior exposure to AI will surely become harder – our study may represent one of the last opportunities to examine developers with limited experience reviewing AI-generated code.

Step C] We used random stratified sampling to divide the participants into Task 1 or Task 2 of the experiment. Stratification was needed to ensure that we assigned an equal share of **AI-devs** and **!AI-devs** to Task 1. The assignment is further described as part of the data collection in Section 3.3.1.

Step D] In Task 1, the **AI-devs** and **!AI-devs** cohorts each added a new feature to an existing Java system. Both the system and the task are described in Appendix A.

The code submitted by these participants is then handed off to another developer in Task 2 for further evolution. Task 1 was concluded by the exit questionnaire presented in Table 2, and the code was analyzed as part of the submission system described in Section 3.3.1. Only one of the Task 1 participants did not submit the questionnaire.

Table 2: Overview of the exit questionnaire. A colon indicates a free-text input field. A letter within parentheses shows a mapping to the corresponding SPACE dimension – items d) and j) are inverted.

Question	**Type**	**Operationalization**
Q2-1. Did you complete the task in one uninterrupted sitting?	Nominal	a) Yes, b) Yes, but with short breaks, c) No
Q2-2. Did you use AI assistants whilst completing the task?	Binary	a) Yes, c) No
Q2-3. (AI-devs only) Which AI assistant did you work with?	Closed list	a) GitHub Copilot, b) Amazon CodeWhisperer, c) JetBrains AI Assistant, d) Visual Studio IntelliCode, e) TabNine, f) ChatGPT, g) Other:
Q2-4. (AI-devs only) How frequently did you interact with the AI assistant?	Ordinal	a) Hardly at all, b) Sometimes, c) Often, d) Almost for every statement I wrote
Q2-5. The task generally resembled development work I have done in the past.	5-point Likert	a) Strongly disagree, b) Disagree, c) Neutral, d) Agree, e) Strongly agree
Q2-6. Please list any development tools used during the task beyond the standard IDE. Examples include code linting tools, quality analyzers, and vulnerability scanners.	Nominal	a) N/A, b) Used tools:
Q2-7. Perceived productivity	5-point Likert + N/A	Thinking of your experience with the task, please indicate your level of agreement with the following statements. a) I was focused on the task during the programming session. (E) b) I was a productive programmer while completing the task. (E) c) I felt fulfilled while completing the programming task. (S) d) I found myself frustrated while completing the programming task. (S) e) I made fast progress despite working with an unfamiliar system. (E) f) The code I wrote was of high quality. (S) g) I maintained a state of flow during the programming task. (E) h) I enjoyed completing this task. (S) i) I completed the repetitive programming activities fast during the task. (E) j) I spent considerable time searching for information or examples during the task. (C)
Q2-8. Is there anything you would like to add regarding the study or your role in the experiment?	Free-text	:
Q2-9. Please provide your email if you want to receive the report when the study is done.	Free-text	:

Step E] In Task 2, which constituted the RCT, new participants were randomly assigned to evolve a valid Task 1 solution. These solutions originated either from the **AI-dev** cohort (treatment) or the **!AI-dev** cohort (control). Note that all Task 2 participants worked without AI assistants, and the task itself is described in Appendix A.2. Finally, Task 2 was concluded with the same exit questionnaire as Task 1 (see Table 2). All but three Task 2 participants submitted the questionnaire.

3.2 Experimental Variables and Hypotheses

We conducted a two-level, single-factor experiment with underlying variations within the two levels. The independent variable was whether participants evolved code that had been co-developed with an AI assistant during Phase 1 (treatment group) or not (control group). As shown in Figure 3, we measured four dependent variables in the Phase 2 RCT.

Completion time was measured by the submission system as the duration between 1) the moment a participant gained access to their unique GitHub repository and 2) the point at which they irrevocably submitted their solution. In line with the definition of technical debt (see Section 2.1), we consider the time it takes to implement changes a valid proxy for maintainability. Between 1) and 2), participants were free to push commits to the GitHub repository as frequently as they liked, allowing them to follow their preferred Git workflow. However, many participants reported not completing the task in one uninterrupted session. For this subset, we largely replaced the measurements with self-reported time estimates. This adjustment is further explained in Section 3.3.1 and critically discussed in Section 6.

CodeHealth (CH) was measured as a project-level weighted average over all files in the codebase, i.e., not only the files modified by the participant. The weighting is based on LoC calculated using *cloc*[13]. CH ranges from 1 to 10 on an interval scale (no true zero) and has been shown to align well with expert assessments of maintainability (Borg et al, 2024a). However, CH has been calibrated over the years for large proprietary systems, where substantial structural problems accumulate over time. To cater to specific needs, CH can be customized by adjusting violation thresholds and the weights of individual rules. In another project, we calibrated CH for smaller development tasks. Together with our partner Codility[14], a company developing programming assessment tools used in recruitment and technical screening, we identified a configuration that better detects fine-grained differences in small code samples – making it a better fit for our study as well. In this custom rule set, two code smells (see Appendix C) have lower thresholds: *Complex Method* triggers at a cyclomatic complexity of 4 (instead of 9), and *Nested Complexity* at three levels of nesting (instead of 4). The configuration file is available in the replication package. Finally, the average CH was calculated in the continuous integration pipeline using GitHub Actions.

Test coverage (TC) was measured as the line coverage of the final solution's test suite, reported as a percentage on an interval scale. We consider TC an indicator of how much attention participants gave to testing. Participants could freely add test cases, and the resulting coverage was measured using the Java code coverage library JaCoCo, integrated into the continuous integration pipeline via the same GitHub Action. We used JaCoCo's default metric, which is line coverage rather than statement coverage.

Perceived productivity (PP) is a subjective assessment based on the SPACE framework (see Section 2.1). We measured PP using a Likert scale composed of ordinal Likert items as part of the exit questionnaire, for which participants received a link upon completing the task. The Likert scale was inspired by the approach described by Ziegler et al (2024) and it allowed us to complement objective measurements with

[13]https://github.com/AlDanial/cloc
[14]https://www.codility.com/

developer perceptions. We adapted the statements to suit both **AI-devs** and **!AI-devs**, and avoided references to specific tools. The scale consisted of ten 5-point Likert items (with an optional N/A response), structured according to the SPACE framework; see Q2-7 a)–j) in Table 2.

Several contextual and individual-level variables inevitably influence outcomes in software engineering experiments. While our RCT helps balance both observed and unobserved confounders across treatment and control groups, the realism introduced by remote participation limits experimental control. In addition to the core variables, we preregistered four contextual variables that we expected would particularly influence the results. Specifically, we measured the following in the pre-screening and exit questionnaires to enable post hoc analysis:

- Whether the participant completed the task in one uninterrupted sitting (Q2-1).
- Which AI assistant was used during Phase 1 (Q2-3).
- The extent to which the AI assistant was used during Phase 1 (Q2-4).
- Whether the participant used additional supportive development tools (Q2-6).

3.2.1 Frequentist Hypotheses

In the frequentist analysis, we hypothesize that the use of AI assistants will have a positive impact on the dependent variables in the Phase 2 RCT. To formally assess these effects, we define four null hypotheses ($H_0 1$–$H_0 4$), each corresponding to one of the dependent variables. Each null hypothesis states that there is *no difference* between participants evolving code (in Task 2) co-developed with an AI assistant and those evolving code written without AI assistance (in Task 1).

$H_0 1$ There is no difference in completion time.
$H_0 2$ There is no difference in CodeHealth (CH).
$H_0 3$ There is no difference in test coverage (TC).
$H_0 4$ There is no difference in perceived productivity (PP).

For each null hypothesis, we also define a corresponding non-directional alternative hypothesis ($H_A 1$–$H_A 4$), stating that there *is* a difference.

3.2.2 Bayesian Causal Analysis

To determine which variables to control for, we rely on causal inference. Figure 4 shows the causal graph underlying our Bayesian analysis. The directed acyclic graph, created using DAGitty, shows how we connect the variables involved in our two-phased study: each vertex represents a variable, and we trace an arrow between two variables when we consider they have a causal relationship. Based on this graph, we can determine which variables need to be controlled for to estimate causal effects.

As an example, consider the effect of using an AI assistant (AI_use) on TC. Developer 1 ($Dev1$) has a level of Java proficiency level ($Dev1\_skill$), which will influence the quality of their submission ($Code1$). In Phase 2, developer 2 ($Dev2$) continues working on this code, and extends it into a new submission ($Code2$), which will have a test coverage (TC). As a result, there is a causal path between AI_use and TC.

Note, however, that in the causal graph, there are two additional paths between AI_use and TC. The previously discussed is *causal* ($AI\_use \rightarrow Code1 \rightarrow Code2 \rightarrow$

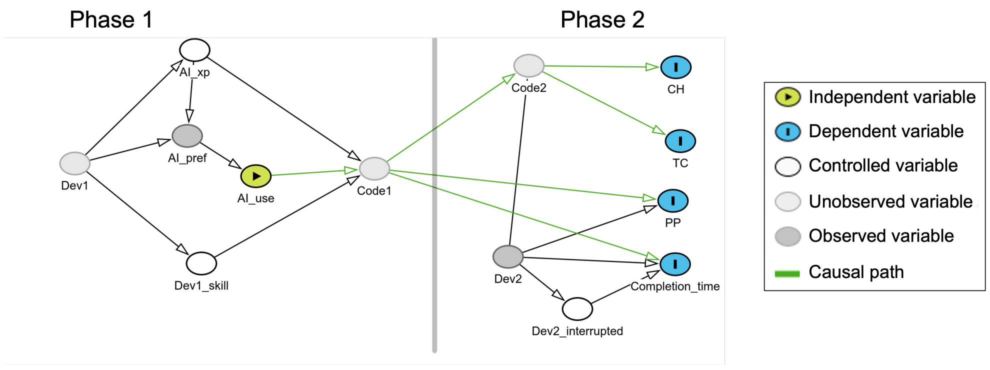


Fig. 4: DAGitty causal graph. *Dev*1 and *Dev*2 represent the full complexity of the human participants in Phases 1 and 2, respectively. *Code*1 and *Code*2 are the participants' solutions after Phases 1 and 2, respectively. *AI_use* is the independent variable. The other variables are explained in Table 3.

TC), since we follow the arrows. However, there are two other paths that connect the two variables, if we ignore the direction of the arrows: ($AI\_use \leftarrow AI\_pref \leftarrow Dev1 \rightarrow Dev1\_skill \rightarrow Code1 \rightarrow Code2 \rightarrow TC$). These paths are called non-causal or backdoor (McElreath, 2020), and indicate the presence of a potential confounder: What if developers who are less proficient at Java prefer to use AI assistants, and more proficient Java developers prefer to avoid them? We would compare two cohorts that are different in *two* ways: More Java-proficient developers who do not want to use AI, and less proficient developers who prefer to use it. As a result, we need to control for the Java proficiency level of developer 1 to isolate the effect of using AI assistants.

Using the graph, we reason in a similar way to deduce that we should also control for developer 1's habitual use of AI assistants. However, there is an important caveat: when developer 1 does not use an AI assistant, their prior habitual use of such tools becomes irrelevant. We present in Appendix B how we model this fact. Throughout the remainder of this paper, we refer to participants who answered "Strongly agree" to the statement "I'm a habitual user of AI assistants while programming" in the prescreening questionnaire (Q1-7a) as "habitual AI users."

Table 3 summarizes the covariates in the causal graph. The last column indicates whether each variable is directly *controlled* for, to block backdoor paths, or *unobserved* (latent). Further details are available in Appendix B.

In summary, we state that *AI_use* (the independent variable), *AI_xp*, and *Dev*1*_skill* causally influence *Code*1 in Phase 1. In Phase 2, *Code*1 and *Dev*2 jointly influence the final solution, *Code*2. By adjusting for *Dev*1*_skill* and *AI_xp*, DAGitty identifies a green causal path that extends through Phase 2 to the four dependent variables. We measure the causal effects on *Code*2 using the two dependent variables CH and TC. The dependent variables PP and Completion time, on the other hand, are causally influenced by both *Code*1 and *Dev*2. When modeling Completion time, we also control for *Dev*2*_interrupted*, to improve the precision of our estimates of other effects, as it has a major impact of the Completion time.

Table 3: Covariates in the causal analysis. The questionnaire is available in Table 1.

Variable	Description	Scale and Operationalization	Role in Analysis
*Dev*1	Full complexity of the Task 1 participant, including intrinsic motivation, cognitive ability, and psychological state.	N/A	Unobserved; assumed balanced by randomization.
*Dev*1*_skill*	Java programming skill of Dev1.	Ordinal 3-point scale, Q1-5.	Controlled; used for adjustment.
AI_xp	Habitual use of AI assistants.	Ordinal 5-point scale, Q1-7a.	Controlled; used for adjustment.
AI_pref	Stated preference for AI-assisted development.	Binary, derived from Q1-7 items b-i.	Used for preceding group assignment (see Section 3.1), not adjusted.
*Code*1	Solution produced by Dev1 in Task 1.	Only partially observed.	Unobserved; treated as latent mediator.
*Dev*2	The full complexity of the human participant in Task 2.	N/A	Unobserved; assumed balanced by randomization.
*Code*2	Solution produced by Dev2 in Task 2.	Only partially observed.	Unobserved; treated as latent mediator.
*Dev*2*_interrupted*	To what extent Dev2 completed Task 2 in one uninterrupted session.	Ordinal 3-point scale, Q2-1.	Controlled; used for adjustment.

Finally, to validate our controlled variables, we examined how the **AI-devs** and **!AI-devs** differ. We report this demographic information in Section 4.1.1, but highlight two key findings here. First, **AI-devs** are more often habitual users of AI assistants, which is expected since we purposefully assigned developers accordingly to encourage compliance. Second, **!AI-devs** include fewer Java beginners and more advanced Java developers. This aligns with previous findings that experienced developers benefit less from using AI assistants (Ziegler et al, 2024; Cui et al, 2025). These two findings validated our controlled variables.

3.3 Data Collection and Validation

This section presents how we collected data during the experiments, and how we validated and corrected it afterwards.

3.3.1 Data Collection

The programming tasks were administered using the *snapcode.review*[15] platform, a service used by Equal Experts for coding tests during recruitment processes. *snapcode.review* automates take-home challenges and integrates with GitHub to manage individual repositories per participant. This solution enabled us to trigger acceptance test runs, run CodeScene analyses (to measure CH), execute JaCoCo measurements (for TC), and collect time stamps (for completion time) as part of a continuous integration pipeline.

We used *Typeform*[16] to distribute and collect both the prescreening and exit questionnaires. Both questionnaires were piloted with four developers from Equal Experts to ensure clarity and relevance. The feedback led to only minor adjustments.

[15]https://snapcode.review
[16]https://www.typeform.com

We opted for a phased rollout of the programming tasks to validate the scalability of our infrastructure. The first batch of participants was recruited through an internal announcement in the Equal Experts developer community on October 10, 2024. Thirty-one developers signed up for the experiment in this round. We assigned eight of them to the Phase 1 **AI-dev** subgroup based on two prescreening criteria. First, participants had to report whether they were habitual users of AI assistants by answering 4 or 5 to statement Q1-7a. Second, participants had to express a preference for working with AI assistants by providing a median response greater than 3 across items Q1-7b to Q1-7i (taking the inverted Q1-7g into account). The remaining participants were randomly assigned to either the **!AI-dev** subgroup or Phase 2. After two weeks, we concluded this pilot phase with satisfactory results and opened up the experiment to all volunteers, using the same assignment criteria.

We invited all qualifying **AI-dev** participants to begin Phase 1 at the end of November, alongside a matching number of randomly selected **!AI-dev** participants. Every time a Task 1 solution was submitted, three new **!AI-dev** participants were continuously invited to build on it as part of Task 2 in Phase 2. We selected three to increase the chances of obtaining at least one valid Task 2 solution per submission.

Both automatic and manual reminder emails were sent to encourage participation. The submission system automatically sent up to two acceptance reminders (after 7 and 12 days) to participants who had not yet accepted their task invitation. Similarly, participants who had accepted, but not yet submitted, received up to two additional reminders. In addition, we manually sent a reminder email to participants approximately one month after they completed the prescreening questionnaire, followed by up to two additional chaser emails. We closed the data collection on January 18, 2025.

3.3.2 Data Validation and Correction

For both tasks, we excluded participants whose solutions failed to pass the acceptance test suite. We manually inspected all passing solutions and their corresponding exit questionnaire responses. We found no indications of attempts to game or manipulate the system, nor any responses suggesting non-serious engagement. However, we identified three protocol violations:

- One **AI-dev** participant (*anon126*) completed Task 1 twice, both times with AI assistance. Since both these solutions received valid follow-up solutions in Task 2, we chose to keep them. While this means that two Task 1 solutions are not independent, it does not affect the RCT in Phase 2. However, it introduces a minor threat to internal validity when comparing **AI-dev** and **!AI-dev** for Task 1, as discussed in Section 6.
- One participant assigned to the **!AI-dev** group self-reported using AI assistance in the exit questionnaire. We resolved this violation by reclassifying the corresponding Task 1 solution as belonging to the **AI-dev** group – after we confirmed in the prescreening questionnaire that this participant's AI experience profile matched the AI-dev group.
- About half of the participants (73 out of 151, 48.3%) self-reported (Q2-1) that they did not complete the task in a single uninterrupted session. As a result, the corresponding submission time stamps recorded by the submission system

are highly unreliable. While randomization mitigates the effects in comparisons between groups, we took additional steps to address this issue, as explained next.

We reached out via email to the participants[17] who responded b) or c) to Q2-1 in Table 2. Moreover, we also reached out to participants who responded a) to Q2-1 but had a recorded completion time of more than 8 hours. We asked all of them to provide their best time estimate with 30 min granularity. This way we received 67 time estimates, which we use instead of the recorded time estimates. The longest estimate we received was 15 hours. Thus, we decided to consider all longer time values ($>$ 16 hours) unreasonable for this study and removed 11 values from the analysis (Task 1: 3 **AI-dev**, 5 **!AI-dev**; Task 2: 1 treatment, 2 control). Note that for the four participants who did not submit exit surveys, two of them had reasonable completion times of less than 5 hours, which we include in the analysis.

We validated the internal consistency of the Likert scale used to measure PP using Cronbach's alpha. After inverting the two negatively worded items, the resulting alpha was 0.82, which shows a good internal consistency. Based on this result, we proceeded to compute mean values for PP rather than medians in the frequentist analysis. In the Bayesian analysis, however, we use the individual Likert items as will be explained in Section 3.4.2.

3.4 Data Analysis

We follow a mixed-methods approach, using multiple methods *"to collect, analyze, and integrate both qualitative and quantitative data in our analysis"* (Storey et al, 2025). Throughout the analysis, we assume independence of observations, i.e., that the participants did not communicate or influence each other. As outlined in the registered report (Borg et al, 2024b), we present both frequentist and Bayesian analyses of the quantitative data. The rationale is that Bayesian analysis will explore uncertainties to provide a robust probabilistic understanding, whereas frequentist hypothesis testing will facilitate communication of results to a broader audience. Moreover, we conduct qualitative analyses of free-text responses from the questionnaires and the submitted source code solutions.

3.4.1 Quantitative Frequentist Analysis

The purpose of the frequentist analysis is to apply inferential statistics to test the hypotheses in Section 3.2.1. We assessed the normality of the dependent variables using Shapiro-Wilk tests. All results are available in the replication package, a summary of the outcomes follows:

- **Completion time.** Normality was rejected. We proceeded with Wilcoxon rank-sum Test and report Cliff's delta as the effect size.
- **CH.** Normality was not rejected, neither was equal variances. We used Welch's t-test and report Cohen's d as the effect size.
- **TC.** Normality was rejected. We proceeded with Wilcoxon rank-sum Test and report Cliff's delta as the effect size.

[17]All but two who explicitly opted out from follow-up questions.

- **PP.** Normality was not rejected, neither was equal variances. We used Welch's t-test and report Cohen's d as the effect size.

Given the limited number of hypotheses, we did not apply corrections for multiple testing. To estimate confidence intervals for the dependent variables, we used non-parametric bootstrapping: drawing 1,000,000 samples with replacement from each group.

As part of our pre-registration, we conducted an a priori power analysis using G*Power (v.3.1.9.7) to determine the required sample size for the Phase 2 RCT. Assuming a medium effect size ($d = 0.5$) for the dependent variables, a significance level of $\alpha = 0.05$, and a desired power of 0.80, the analysis indicated a minimum of 128 participants to support frequentist hypothesis testing. As our study design assigns 50% of participants to Phase 1, this implies a target of 256 completed tasks across both phases. As reported in Section 3.1, 449 participants signed up and 151 participants completed their tasks (33.6%), i.e., they submitted a solution that passed the corresponding acceptance tests. This means we did not reach our target.

As will be reported in Section 4, we had a total of 75 Phase 2 participants. For this sample size, two-sided tests at $\alpha = 0.05$ achieve 80% power for standardized mean differences of approximately $d \approx 0.66$. For a medium effect size ($d \approx 0.5$), the power is approximately 0.57, and for a small effect ($d \approx 0.2$) approximately 0.17. Hence, the Phase 2 frequentist tests are well powered to detect large effects of prior AI assistance on Task 2 outcomes, but remain underpowered for small to moderate effects. This further motivates the Bayesian complement.

3.4.2 Quantitative Bayesian Analysis

We conducted a Bayesian analysis to gain a probabilistic understanding of the treatment effects, an approach that remains robust even in the case of smaller sample sizes (McElreath, 2020).

Statistical Models

Our choice of statistical models is guided by the type of variables we consider, as dependent and independent variables (also called predictors). Throughout this section, we indicate dependent variables (defined in Section 3.2) in **bold** and predictors (defined in Section 3.2.2) in *italics*. *AI_xp*, *Dev1_skill*, and *Dev2_interrupted* are ordinal predictors. *AI_xp* ranges on a five-point scale based on habitual use of AI assistants (Q1-7a). *Dev1_skill* refers to the self-reported (Q1-5) Java proficiency using the levels: Beginner, Intermediate, and Advanced. *Dev2_interrupted* is developer 2's answer to Q2-1, i.e., whether Task 2 was completed in one uninterrupted sitting: Yes, Short breaks, and No.

Completion time is a metric (continuous) variable, strictly positive. **CH** is a metric variable, on a scale between 1 and 10. **TC** is a continuous variable on a scale between 0 and 1 (a percentage). **PP** is measured using a Likert scale in the exit questionnaire. Unlike the frequentist analysis, which uses the mean value, the Bayesian analysis uses the individual Likert items to estimate a latent variable.

Since *AI_xp*, *Dev1_skill* and *Dev2_interrupted* represent ordered categories, we expect, for example, that a developer with an advanced level of Java also has the level

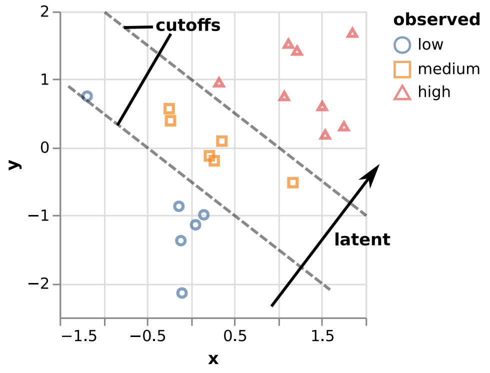


Fig. 5: A diagram explaining an ordered logistic regression model. If we observe the two predictors x and y, as well as the observed responses (*observed*). The model infers both a latent score (*latent*) for each point (as a linear combination of x and y), and cutoffs between the different response levels. In other words, the model finds hyperplanes separating each response level with the next, where all hyperplanes are parallel.

of a beginner. Modeling the effects of these variables as nominal or continuous would be incorrect (Bürkner and Charpentier, 2020). We therefore use an appropriate model, described in more detail in Appendix B.

For **CH**, we use a classical linear regression with normally distributed residuals. We choose this model because the CH score for a solution is the weighted average of each file's scores. Because of the central limit theorem, we would expect residuals to be normally distributed.

For **TC**, we use a fractional logistic regression model, since the variable is a percentage bounded between 0 and 1. In essence, this model is a classical linear regression on the logit of the percentage (Papke and Wooldridge, 1996). We also tested with a classical linear regression, with similar results, since the range of test coverages in our dataset is narrow (65% to 75%).

For **PP**, we use a more complex model for two reasons. First, each answer is obtained using a set of ten discrete, ordinal Likert items (questions) on a narrow range (1-5). Second, and more importantly, PP is not measured *directly*, but instead approximated via these 10 questions. It is possible that some questions are more relevant than others when estimating a developer's productivity. To model this type of answer, we use an ordinal logistic regression (McElreath, 2020; Gelman et al, 2021), with a latent variable representing PP.

Figure 5 displays a simple example of an ordered logistic regression. In our case, the model represents the productivity of each developer using a latent PP score, predicted from the independent variables. This latent score is then reflected in the answers to each question, where each question has a different set of cutoffs.

This model has three major advantages: First, it estimates the properties of each question, i.e., the cutoffs between response levels. Second, it uses information from every question in estimating the developer productivity, making our estimates of the developer's productivity more precise. This estimate is then what we use as a response variable when studying the effect of predictors on PP.

Because we work in a Bayesian inference setting, our estimates of developer productivity are imprecise, which is reflected in the posterior distribution of productivity. This uncertainty is taken into account when we estimate the effect of predictors too.

Sensitivity Analysis

In a Bayesian setting, the priors of effects at play can influence the analysis. To study the importance of priors, we constructed three types: i) uninformative priors (neutral), ii) AI-skeptical priors (pessimistic), and iii) AI-enthusiastic priors (optimistic), to reflect different stances on the maintainability impact of AI assistance — this serves as a sensitivity analysis.

For **Completion Time**, we used an optimistic prior based on GitHub's research (Peng et al, 2023), which reported a 55% improvement. While the GitHub research investigated the direct effects of working with AI, we use the same improvement to inform our optimistic prior for subsequent manual development. For the other response variables, we defined priors that implied an effect of about one standard deviation in the response, where the optimistic prior predicted an improvement, and the pessimistic prior a decline. Table 4 summarizes our priors for the effect of AI. For full details about our statistical models, we refer to Appendix B.

Table 4: Summary of priors used in the Bayesian Analysis.

Variable	**Prior Type**	**Distribution (Normal)**
Completion Time	Optimistic	$\mathcal{N}(-0.55, 0.3^2)$
	Neutral	$\mathcal{N}(0.0, 1.0^2)$
	Pessimistic	$\mathcal{N}(0.55, 0.3^2)$
Test Coverage	Optimistic	$\mathcal{N}(0.25, 0.5^2)$
	Neutral	$\mathcal{N}(0.0, 1.0^2)$
	Pessimistic	$\mathcal{N}(-0.25, 0.5^2)$
CodeHealth	Optimistic	$\mathcal{N}(0.5, 0.5^2)$
	Neutral	$\mathcal{N}(0.0, 1.0^2)$
	Pessimistic	$\mathcal{N}(-0.5, 0.5^2)$
Perceived Productivity	Optimistic	$\mathcal{N}(0.25, 0.5^2)$
	Neutral	$\mathcal{N}(0.0, 1.0^2)$
	Pessimistic	$\mathcal{N}(-0.25, 0.5^2)$

Reported Statistics

For each model, we report the influence of a Task 1 participant using AI, compared to another Task 1 participant of the same skill. Now, the effect of using AI presents an *interaction* with AI experience: If the participant has more experience with AI ($AI\_xp = 5$), i.e., a habitual AI user as defined in Section 3.2.2, the effect of using AI can be expected to be more important. We focus our reporting on the effects of evolving habitual AI users' code, but this does not mean that we excluded non-habitual AI users from the analysis. They are part of the data, but our Bayesian model assumes that AI experience has a monotonic effect, so for developers with lower experience, the effect of using AI is expected to be closer to zero, in comparison (see Appendix B for the full model specification).

Besides the effect of AI, we also report the effect of higher skill on the outcome. We define "higher skill" as the difference between outcomes for developers with $Dev1\_skill = 3$ vs developers with $Dev1\_skill = 1$, keeping all other factors equal.

For each estimate, we report the probability of direction ($P(\Delta > 0)$ or $P(\Delta < 0)$), i.e., the posterior probability that the parameter is strictly above (or below) zero. When the probability of direction exceeds 95%, we consider the effect to be significant. This probability is somewhat similar to p-values in frequentist statistics. We also provide 95% Credible Intervals (CrI), which are the Bayesian equivalent of confidence intervals. When relevant, we plot the posterior distribution, or perform posterior predictive simulations to compare the Task 2 results one might expect when $Dev1$ uses an AI assistant versus when $Dev1$ does not.

3.4.3 Qualitative Analysis

This section describes how we conducted qualitative analyses of Free-text Responses and Source Code, respectively.

Free-text Responses

We collected qualitative feedback from participants through free-text responses in the exit survey. In total, we received 59 responses from Phase 1 and 45 from Phase 2. Additionally, more than a third of the participants shared insights via (often rich) follow-up emails, typically in response to clarification questions we sent after task completion. Despite conducting the experiment remotely, we believe we gained a solid understanding of participants' perceptions.

The first and sixth author started by independently conducting inductive coding of the free-text responses. Already at this stage, we observed strong agreement on high-level themes, supported by the participants' focused and relevant feedback. Notably, there were no junk responses. The two authors discussed discrepancies and collaboratively refined the initial coding scheme into a two-level structure. The first author then re-applied the coding, and the sixth author validated the outcome. The final coding scheme is presented in Table 5, organized around three high-level topics 1) Development without AI, 2) AI-assisted development, and 3) Task reflections. The number of times the codes were applied is also reported, which will be discussed in Section 4.5.

Source Code

We also spent substantial effort in qualitatively analyzing the submitted source code. For Task 1, we applied a systematic approach to investigate the impact of AI assistants. We began by sampling five random **AI-dev** and five random **!AI-dev** solutions. The first, third, fourth, and sixth author independently analyzed them to identify interesting variation points. We then met to discuss our findings and compiled a list of 15 aspects that appeared to characterize the solutions. We organized them into four categories: 1) refactoring, 2) solution style, 3) testing, and 4) misc. Furthermore, we found it useful to describe a) the overall solution type, and b) specific changes to the search method presented in Figure 23.

Table 5: Coding scheme for free-text responses. Numbers in parentheses show the number of occurrences.

1) Development without AI (30)	
Refactoring (9)	Manual restructuring of code to improve readability, design, or maintainability.
Testing and debugging (5)	Manually creating or running tests, identifying and reproducing defects, and fixing them.
Learning and onboarding (7)	Understanding of the tech stack or onboarding into the specific project without AI support.
Misc. (9)	Other development activities.
2) AI-assisted development (46)	
Refactoring (4)	Using AI to suggest or implement code improvements or restructuring.
Testing and Debugging (4)	Leveraging AI to create tests, identify and reproduce defects, and assist in fixing them.
Learning and onboarding (8)	Using AI to explore unfamiliar technologies, frameworks, or project-specific structures.
Productivity boost (5)	Reports of increased speed or efficiency as a result of AI assistance.
AI limitations (10)	Situations where AI failed, gave incorrect suggestions, or hindered progress.
Interaction mode (10)	Descriptions of how participants engaged with AI.
Misc. (5)	Other development activities.
3) Task reflections (101)	
Instruction uncertainty (9)	Confusion or ambiguity related to task requirements or scope.
Tech stack (28)	Comments on familiarity or challenges with the programming tools or frameworks.
Setup friction (25)	Difficulties encountered during environment setup, configuration, or initial project use.
Quality expectations (21)	Uncertainty or reflections about how much to improve, test, or refactor the code.
Simplistic task (18)	Perceptions that the assignment was too easy, trivial, or lacked meaningful complexity.

Following this joint meeting, we independently re-coded the same sample of Task 1 solutions and found satisfactory inter-rater agreement. To scale the analysis, we designed regular expressions to automatically identify keywords indicative of specific attributes of Task 1 development. Running these expressions over the `git diff` output between each modified and original repository enabled us to efficiently identify the addition of unit tests (e.g., assertions) or comments, as well as code patterns suggesting a shift to the functional paradigm (e.g., map, collect, and reduce).

To complement this manual analysis, we also ran detailed static analysis of the Task 1 solutions using CodeScene (v6.19.15) and PMD (v7.13.0 with default settings) to identify patterns by comparing with the base *RecipeFinder* (see Appendix A.3). Moreover, we ran RefactoringMiner 2.0 by Tsantalis et al (2022) on all Task 1 `git diffs` to investigate what types of refactoring operations the participants undertook.

4 Results

We organize this section as follows. First, we present participant demographics and descriptive statistics related to Tasks 1 and 2. Second, we report the results from the preregistered RCT, organized per RQ. Third, we present observational findings on the use of AI assistants in Task 1, which provide context for interpreting our findings. Finally, we summarize insights from the qualitative analysis of free-text answers to add additional nuance to the quantitative results.

Throughout this section, we complement the quantitative findings with quotes from participants. The quotes add qualitative nuance and constitute either representative examples of a theme or contrasting perspectives. All mentioned participants are professional developers unless otherwise stated. Shorter quotes are inlined, longer quotes are separated and indicated with a blue vertical line. When presenting longer quotes, we report the following metadata (the four dependent variables listed in parentheses):

- Participant ID, Task {1, 2}, {**AI-dev**/**!AI-dev**} [HABITUAL] (if Q1-7a=5) or {Treatment/Control} , Role (Q1-4)+Java proficiency (Q1-5), AI assistants used (if applicable), resemblance agreement (R1-5, Q2-5), added LoC, (*time*, *CH*, *TC*, and *PP*).

4.1 Descriptive Statistics

In total, 449 participants signed up for the experiment by completing a valid prescreening survey. While controlled experiments in software engineering tend to largely rely on student participants (Feldt et al, 2018), this study stands out: 92.2% of our participants were professional developers. The median age group was 40-49 years, underscoring the experience level of our volunteer base.

The prescreening revealed that 73.6% of the volunteers had prior experience with AI assistants (Q1-6 in Table 1). We asked this subset to self-report their experience using the Likert scale in Q1-7, with results presented in Figure 6. These responses provide a timely snapshot of AI-related development experience and preferences at the end of 2024 – a very fast-moving area of practice.

We assigned the 118 participants who expressed a preference for AI-assisted development to the **AI-dev** group in Task 1. Among the remaining participants, 106 were randomly selected for the **!AI** group in Task 1. The remaining 225 participants were reserved for the RCT in Task 2. The remainder of this section presents demographics and task-level summary statistics.

4.1.1 Task 1 Demographics

Task 1 was designed to establish a baseline for the RCT in Task 2. The task concluded with 39 valid **AI-dev** solutions and 37 **!AI-dev** solutions. We consider this a solid starting point for the RCT. More specifically, the Task 1 completion rates were as follows:

- **AI-devs** submitted 38 valid solutions (38 of 118, 32.2%) and a corresponding exit questionnaire. Sixteen of them (42.1%) were habitual AI users. For the remaining 80 participants
  - 51 participants never started the task,
  - 20 participants never submitted a valid solution, and
  - 9 participants actively dropped out.

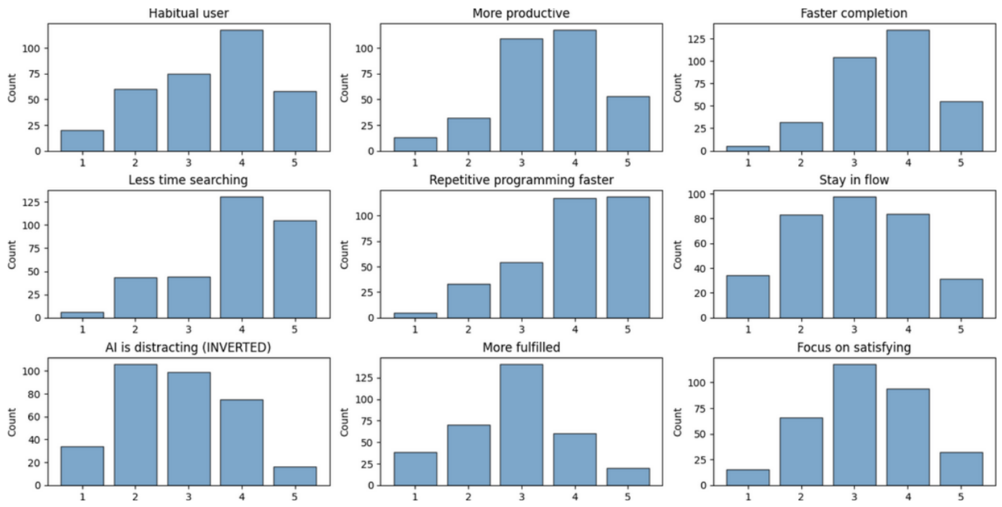


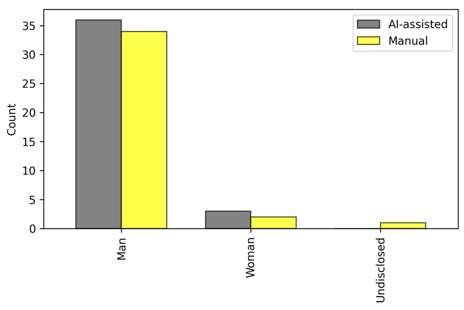
Fig. 6: Respondents' experience and preference with AI assistants (Q1-7 in Table 1), 1=Strongly disagree, 5=Strongly agree.

- **!AI-devs** submitted 38 valid solutions (38 of 106, 35.8%) – three of them never submitted the exit questionnaire, but are still included. As reported under protocol violations in Section 3.3.2, one participant reported using an AI assistant and was thus moved to **AI-dev**, which explains the 39/37 split. Note that this person was not a habitual AI user. For the remaining 67 participants
  - 36 participants never started the task,
  - 23 participants never submitted a valid solution, and
  - 8 participants actively dropped out.

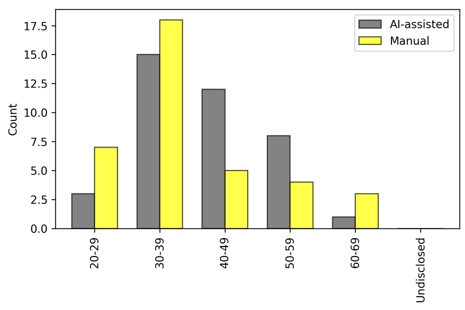
Figure 7 shows demographics for the Task 1 participants. We found that most participants were men and between 30-49 years old. Most participants were professional developers, but there were slightly more Java beginners in the **AI-dev** group. This aspect will be further analyzed later. Overall, we do not observe any major systematic biases that threaten the validity of the RCT.

Figure 8 shows the geographic distribution of Task 1 participants. We notice participation from many Western countries. However, we find that neither South America nor Africa is represented. Notably, several major countries with large developer populations, including China, Japan, and South Korea, are also absent.

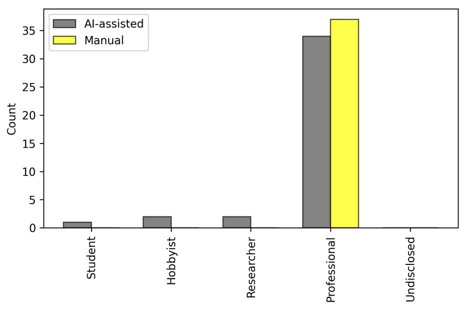
Table 6 shows the frequencies of AI tool usage reported by **AI-devs** in Task 1 (Q2-3), both for all participants and habitual AI users only. The bolded alternatives were explicitly listed in the questionnaire, whereas all others were entered manually by participants. We note that GitHub Copilot was clearly the most used AI assistant, followed by ChatGPT and Cursor. We observed no instances of fully autonomous agents being tasked with the entire problem in Task 1, although recent versions of Cline and Claude can be used in such workflows. Still, some participants relied heavily on autonomous code generation for parts of the task – shifting into product managers



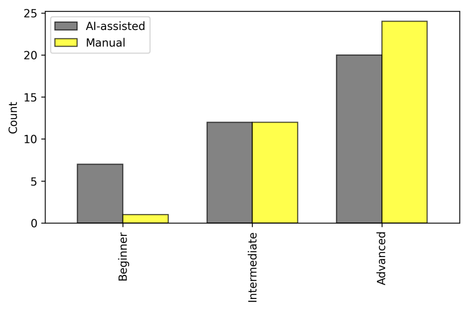
(a) Gender



(b) Age



(c) Role



(d) Java Skill Level

Fig. 7: Demographic distribution of Task 1 participants.

and treating the AI assistant as a junior developer. Regarding the frequency of AI usage (Q2-4), we obtained: Every statement 17, Often 10, and Sometimes 12.

Table 6: Frequencies of AI tools reported in Task 1 (Q2-3).

AI tool	**GitHub Copilot**	**ChatGPT**	Cursor	**JetBrains AI**	Claude	Tailwind AI	**VS IntelliCode**	Mixtral	Microsoft Copilot	Windsurf	Cline	Grok	Gemini	Supermaven	**CodeWhisperer**	**TabNine**
All participants	21	13	9	5	4	2	2	1	1	1	1	1	1	1	0	0
Habitual AI users	8	7	7	1	0	2	0	0	0	1	0	0	1	0	0	0

Some Task 1 participants reported using (non-AI) development tools beyond a standard IDE. In **AI-dev**, three used SonarQube and one used Docker. In **!AI-dev**, the following tools were each reported once: SonarQube, GitHub Codespaces, Firefox,

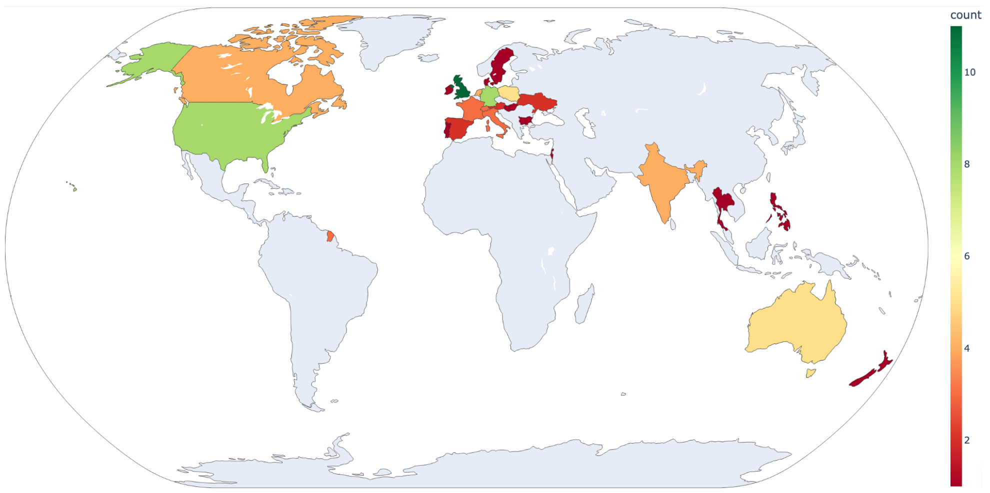


Fig. 8: Location of Task 1 participants.

and Chrome. The two web browsers' debug windows were used to check the web application. The four participants using SonarQube submitted solutions with CH 8.35, 8.34, 8.23, 8.16, respectively. These scores do not stand out (see Figure 21b), and we consider the influence of additional development tools to be negligible and thus controlled in the rest of the analysis.

4.1.2 Task 2 Demographics

Task 2 involved extending the feature developed by an unknown Task 1 developer. In the end, 75 out of 225 (33.3%) assigned participants submitted valid Task 2 solutions. The completion rate matches what we observed for both **AI-dev** and **!AI-dev** groups in Task 1. Figure 9 shows an overview of how the Task 1 solutions were evolved by Task 2 participants. The number of Task 2 submissions building on Task 1 solutions varied between 0 and 4[18]. Of the 75 Task 1 solutions, 24 were not evolved further (see the black box in Figure 9). Note that these 24 solutions had CH scores comparable to the 51 that were evolved (mean 8.32 vs. 8.35; median 8.26 vs. 8.34). The most common outcome was that a Task 1 solution was evolved by one Task 2 participant. In some cases, individual Task 1 solutions were evolved by multiple Task 2 participants. More specifically, 14 Task 1 solutions were each used as the starting point for two different Task 2 participants, 3 solutions for three different participants, and 1 solution for four different participants.

Figure 10 shows the demographics of the 75 participants who completed Task 2. As in Task 1, most participants were men, and the median age in both groups was 40-49 years. Similarly, the Task 2 participants are predominantly professional developers. The reported levels of Java mastery differed slightly between the treatment and control groups, which we account for later.

[18]On a few occasions, we assigned more than three participants to a single Task 1 solution – essentially due to rounding in the assignment logic.

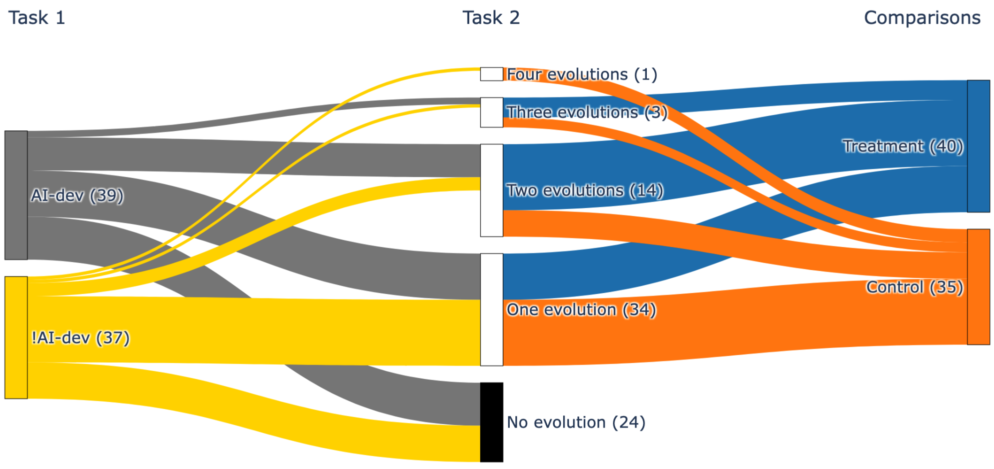


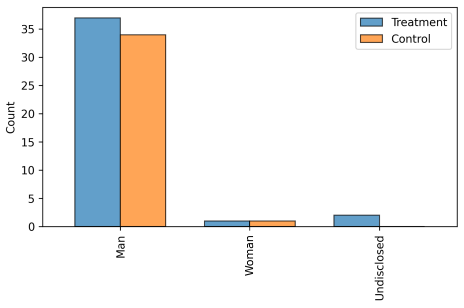
Fig. 9: Flow of Task 1 solutions into Task 2 evolutions by experimental group. The treatment group evolves solutions originally completed with AI assistance (AI-dev), whereas the control group evolves solutions completed without AI assistance (!AI-dev).

Figure 11 shows the geographic distribution of Task 2 participants. The distribution resembles what we found for Task 1, but now both South America and Africa are represented. However, there are still no participants from China, Japan, or South Korea.

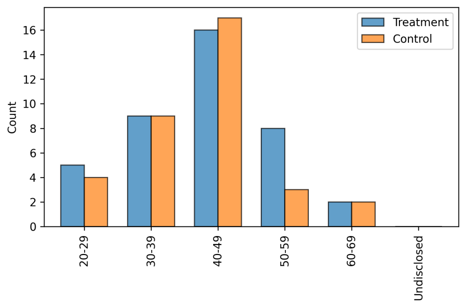
Eight Task 2 participants reported using development tools beyond a standard IDE. In the treatment group, three used SonarQube and one used GitHub Codespaces. In the control group, three used SonarQube and one used Postman. The six participants who used SonarQube submitted solutions with CH 8.84, 8.55, 8.54, 8.32, 8.25, and 8.24, respectively. The highest score (8.84) was obtained by the ambitious *anon050*, who spent 5.5 hours on the task and also achieved the top TC score (90%). The two lowest scores belong to 1) a Java beginner who stated *"I found it fun working in an unfamiliar language. My main language is C# and I work normally in Visual Studio and with .NET."* and 2) an advanced developer who complained about the lack of unit tests: *"I'm very used to TDD so finding no to very few tests, specially unit tests, is somewhat frustrating."* The remaining three CH scores align well with the overall distribution (see Figure 18), and we find no indications of systematic differences caused by the tool usage.

4.1.3 Task-Level Summary Statistics

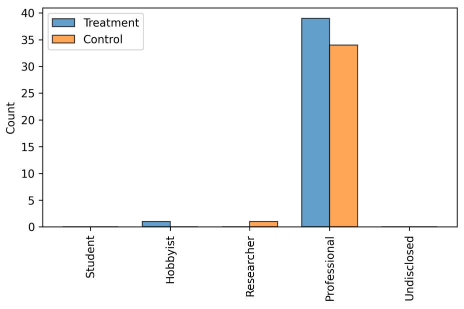
Table 7 shows summary statistics of git activity in Task 1 and Task 2. In Task 1, we notice that four **AI-devs** skew the distribution by adding more than 2,000 lines of code – well above the maximum number of 660 observed in the **!AI-dev** group. Still, the median number of added lines for **!AI-dev** is lower than for **AI-dev**. The four most prolific **AI-devs** were:



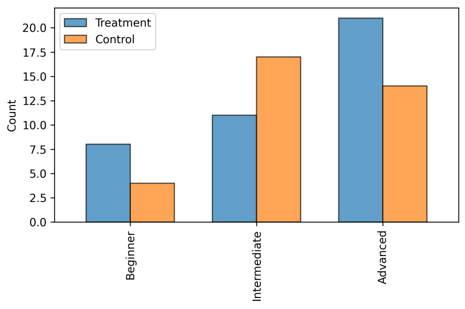
(a) Gender



(b) Age



(c) Role



(d) Java Skill Level

Fig. 10: Demographic distribution of Task 2 participants.

anon126 Java beginner. Used Cursor and Tailwind AI. Also the participant who completed Task 1 twice (see Section 3.3.2). In the first submission, *anon126* explained pushing AI to the maximum, resulting in 10,788 added lines. Despite the volume, the CH was the second highest in Task 1: 8.88. Note that the added lines did not include extra features beyond the task; roughly 90% consisted of Javadoc comments, test code, and a large Tailwind library.

anon054 Advanced Java developer. Used Cursor, ChatGPT, and Cline. Added 3,094 lines, and achieved the highest recorded CH: 9.12 with 98% TC. Explained:

> "*My approach was to learn how well it works letting the AI do most of the work on its own and me being the mentor like with a new colleague at work steering where necessary. [...] Cline did most of the work autonomously based on the existing code and description.*" — `anon054, Task 1, Pro+Advanced, AI-dev [HABITUAL], ChatGPT+Cline+Cursor, R4, 3094 LoC, (5h, CH=9.12, TC=98%, PP=4.4)`

anon139 Java beginner. Used GitHub Copilot, ChatGPT, VS IntelliCode, and Claude. Added 2,032 lines. Explained *"going a bit deep in making it more maintainable"*, but the CH remained at a modest 8.48.

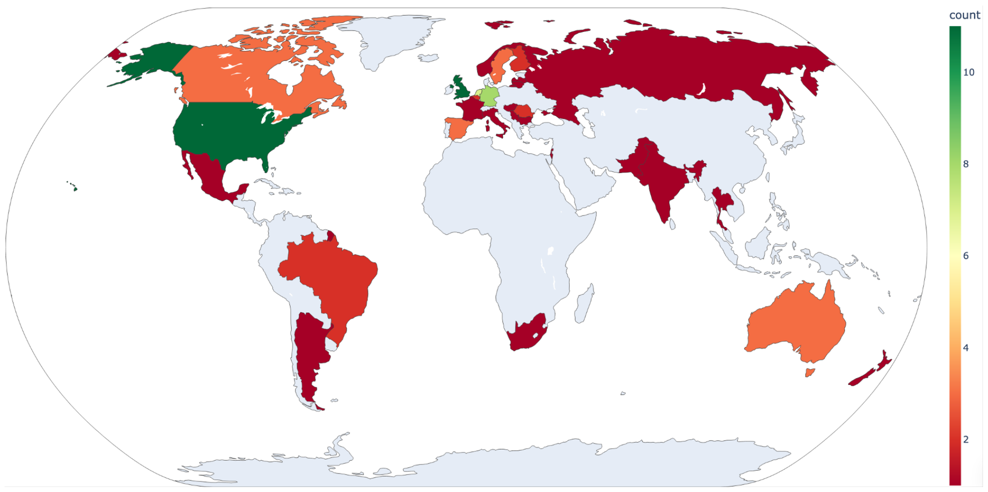


Fig. 11: Location of Task 2 participants.

anon073 Advanced Java developer. Used Cursor. Added 2,635 lines and reached CH 8.38. Provided a rich comment:

> *"I haven't programmed in Java for 15 years, and never touched Spring Boot. It would have taken me a lot longer without the help of an AI. The interaction with the AI felt like pair programming with someone with a vast experience but lazy who sometimes gives hacky answers. I had to constantly remind the AI to use the most recent and best practice information to get better code. I do have 30 years experience in development and multiple frameworks. I felt that that experience was important with identifying subpar code or dead leads that the AI suggested. Just pointing out those to the AI was enough for it to come up with better code. I hardly had to tamper with the code manually."* — `anon073, Task 1, AI-dev [HABITUAL], Pro+Advanced, Cursor, R4, 2635 LoC, (4.5h, CH=8.38, TC=67%, PP=4.2)`

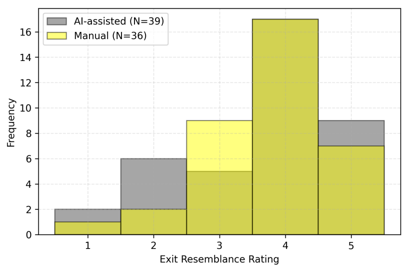
We note that the two Task 1 participants who added the most code using AI also obtained the highest CH. This suggests that heavy use of AI assistants can result in code that is, on average, easily understandable by humans. However, maintainability effort also increase with the overall size of the codebase. We revisit this dilemma in Task 2.

The Task 2 git activity of the treatment and control groups (see Table 7) appears similar. The median number of added and deleted lines is slightly higher in the control group, but the difference is modest. We have no other patterns to report.

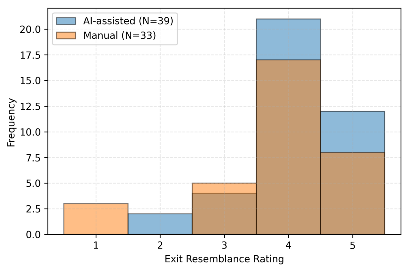
Figure 12 shows answers to Q2-5, i.e., the level of agreement to the statement "The task generally resembled development work I have done in the past." The results show that both Tasks 1 and 2 were perceived as realistic development tasks, which supports the validity of our study.

Table 7: Descriptive statistics of git activity.

	Task 1		Task 2	
	AI-dev (n=39)	!AI-dev (n=37)	Treatment (n=40)	Control (n=35)
commits				
mean / std	6.85 / 13.66	5.16 / 7.48	5.62 / 12.16	4.40 / 4.72
min / median / max	1 / 3 / 82	1 / 3 / 38	1 / 2 / 74	1 / 3 / 21
added lines				
mean / std	642.74 / 1801.63	216.41 / 187.41	158.62 / 202.54	154.23 / 172.97
min / median / max	18 / 123 / 10,788	22 / 166 / 660	6 / 56.5 / 703	7 / 80 / 703
deleted lines				
mean / std	128.54 / 141.45	102.32 / 111.79	72.53 / 129.67	103.26 / 196.53
min / median / max	8 / 74 / 642	4 / 57 / 512	0 / 10 / 562	0 / 17 / 854
changed files				
mean / std	10.92 / 14.92	9.65 / 6.71	5.92 / 5.90	6.60 / 6.04
min / median / max	2 / 7 / 88	2 / 8 / 29	1 / 4 / 29	2 / 4 / 27



(a) Task 1.



(b) Task 2.

Fig. 12: Perceived resemblance to prior development work (Q2-5: 1=Strongly disagree, 5=strongly agree).

4.2 RQ1: More Efficient Manual Evolution?

This section reports results related to the two dependent variables Completion time and Perceived Productivity (PP) (see Figure 2). For each variable, we discuss the distribution followed by frequentist inferential statistics and a Bayesian analysis. We conclude by summarizing the effect of the independent variable, expressed as treatment minus control, in a textbox with the following structure:

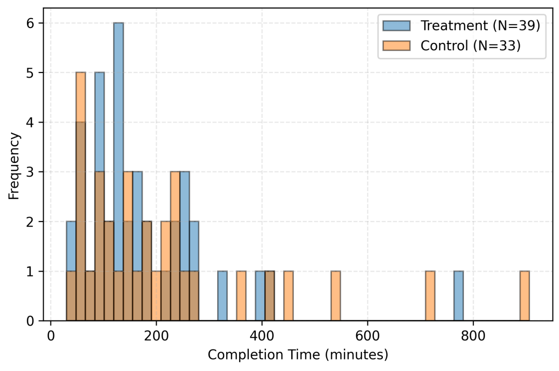
- Frequentist analysis: mean/median difference; 95% confidence interval for the difference; p-value; effect size.
- Bayesian analysis: Probability of direction (positive/negative); posterior mean percentage change; 95% credible interval for the change.
- A brief textual synthesis of these statistics.

4.2.1 Completion Time

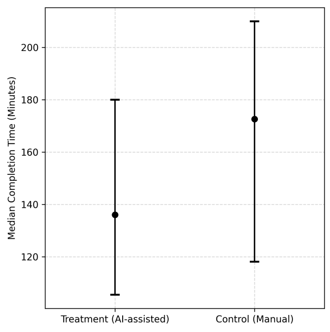
Figure 13a shows the distribution of 72 completion times for Task 2. Most participants completed the task within 300 minutes, and as described in Section 3.4.1, the data is not normally distributed. The longest time was 13 hours, based on an estimated breakdown from one particularly ambitious participant, an advanced Java developer, who explained their process as follows:

> *"I'd say I've spent around 12-16 hours studying the stack – last time I worked with Java it was Java 7, and a lot has changed. In that time I did, of course, spend some time experimenting with different solutions which did not make it into the project. After settling with the JPA Specification API, it took me 2-4 hours to have the acceptance test green [...] Sticking to the role I was playing as a consultant, I then invested some time in refactoring, which then took another 8-12 hours, I'd say. I wasn't sure if this was expected for the task, but its description gave me the impression that the assessment would be reviewed as if I were hired to do a job."* — `anon123, Task 2, Treatment, Pro+Advanced, R5, 645 LoC, (13h, CH=8.89, TC=71%, PP=2.7)`

We excluded the initial 12-16 hours of exploratory learning and summed the remaining estimates to 13 hours. This is potentially an underestimate, and it shows that some participants were highly motivated to deliver high-quality solutions.



(a) Distribution of completion times.



(b) 95% confidence intervals.

Fig. 13: Task 2 completion time.

Frequentist Analysis

Figure 18b shows 95% confidence intervals for the median completion times. The treatment group had a median of 136 minutes (95% CI: 105.5–180.0), while the control group had a median of 173 minutes (95% CI: 118.1–210.0). Cliff's δ was -0.079, indicating a *negligible effect size*. The difference was *not statistically significant* (Wilcoxon rank-sum test, p=0.56).

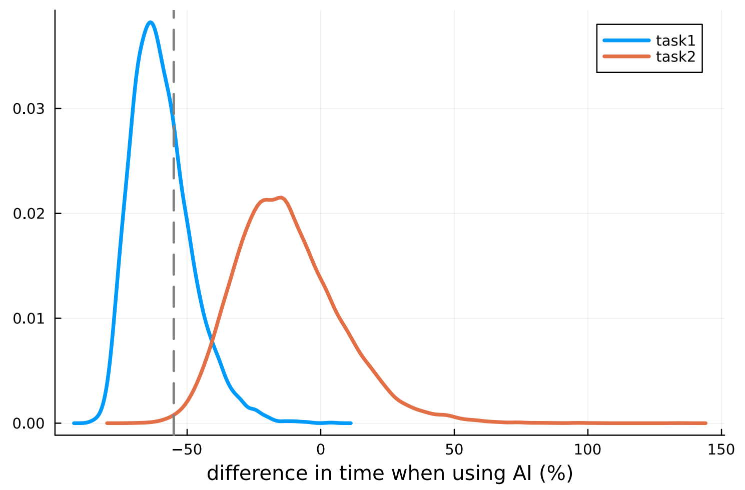


Fig. 14: Posterior effect on completion time when a habitual AI user applies AI assistants in Task 1. The orange curve shows the effect on manual evolution in Task 2, i.e., the focus of our study.

Bayesian Analysis

For Task 1, our Bayesian model shows that using an AI assistant reliably reduced completion time ($P(\Delta < 0) > 99\%$). Figure 14 shows the posterior distribution of the treatment effect on completion time when the $Dev1$ was a habitual AI user. AI-assisted babitual AI users finished Task 1 approximately 60% faster than **!AI-devs** on average (CrI: $[-77.11\%, -30.64\%]$). Our results are consistent with those of Peng et al (2023), indicated by the dashed line in Figure 14.

As a point of comparison, changing $Dev1$'s skill from "Beginner" to "Advanced" had a non-significant effect ($P(\Delta < 0) = 83\%$), with a mean effect of -21.4% (CrI: $[-60.46\%, +34.28\%]$).

For Task 2, however, the positive effect of prior AI assistance observed for habitual AI users largely vanishes. We find no significant effect on $Dev2$'s completion time ($P(\Delta < 0) = 76\%$). The posterior mean effect of using AI is -12.59%, but the posterior distribution is wide (CrI: $[-45.77\%, +32.68\%]$), encompassing both substantial speedups and slowdowns. The effect of higher $Dev1$'s skill is also not significant: the mean effect is -27.89%, (CrI: $[-57.98\%, +12.72\%]$).

Moreover, the posterior probability that prior use of AI made $Dev2$ faster is 76%, but this estimate is sensitive to the choice of priors. When using a pessimistic prior, the posterior probability that prior AI use decreases Task 2 completion time fell to about 38-39%, which shows that we do not have conclusive results yet.

In contrast, a major influence on Task 2 completion time is whether $Dev2$ completed the task in one uninterrupted session or not (see $Dev2\_interrupted$ in Figure 4). When $Dev2$ took breaks (interruption levels 2 or 3), they finished 173.66% later than when they did not (CrI: $[+37.69\%, +416.88\%]$).

Quantitative Results – Task 2 Completion Time (Trt-Ctrl)

- **Frequentist.** Median difference = −37 minutes; 95% CI [−90, +38]; Wilcoxon $p = 0.56$; Cliff's $\delta = -0.079$.
- **Bayesian.** When Task 1 was completed by a habitual AI user: $P(\Delta < 0)$ = 76%; Posterior mean change = −12.6%; 95% CrI [−45%, +33%].
- Both analyses are consistent with a null effect on Task 2 completion time.

4.2.2 Perceived Productivity

Figure 15a shows the distribution of PP for 72 Task 2 solutions. As described in Section 3.4.1, we consider the data approximately normally distributed. We continue by describing the seven participants who stand out in Figure 15a, namely four with the highest PP scores and three with the lowest. All of them agreed or strongly agreed that the task resembled previous development work (Q2-5). Three of them reported low PP (2.7):

anon123 This participant, discussed in Section 4.2.1 as the one who spent the most time, evolved **AI-dev** and made substantial changes, i.e., 14 files in 16 commits. The low PP is explained in Q2-8:

> "*It's been years since I've touched any Java code, even longer since using Spring [Boot]. The broadness of the task also left it open to which optimizations should be done, but I feel much more satisfied and know the system is easier to work with after working on it*" — `anon123, Task 2, Treatment, Pro+Advanced, R5, 645 LoC, (13h, CH=8.89, TC=71%, PP=2.7)`

anon014 A beginner Java developer who spent 12 hours on the task, evolving **!AI-dev** code. Despite the long time, only four files were changed across three commits. The low PP is attributed to the participant's inexperience with Java web apps:

> "*I also spent about 6 additional hours on non-essential environment setup tasks (i.e. Containerizing the app so I didn't have to run a JDK locally). In case this is relevant, I am not primarily a Java developer, and this was my first time working with a Java web application.*" — `anon014, Task 2, Control, Pro+Beginner, R4, 33 LoC, (12h, CH=8.33, TC=70%, PP=2.7)`

anon119 An advanced Java developer who spent 134 minutes on the task and evolved **AI-dev** code. Surprisingly, the participant made many changes in a short time, i.e., 7 files in 20 commits, but still felt unproductive.

Four participants, all advanced Java developers, reported a PP of either 5 or 4.9.

anon100 Evolved **AI-dev** code. Worked for 4.5 hours, changing 6 files in 21 commits.

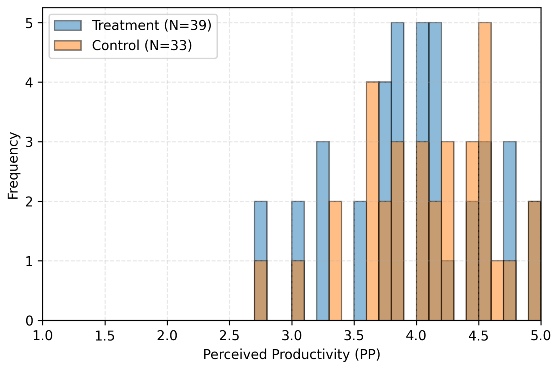
> *"I adhered to the Pomodoro technique. For around every 24 minutes, I'd take a 6-minute break. I'd estimate I spent 3.5 hours on task typing."* — `anon100, Task 2, Treatment, Pro+Advanced, R5, 290 LoC, (4.5h, CH=8.61, TC=69%, PP=5.0)`

anon109 Evolved **AI-dev** code. Solved the task in a single commit with minimal churn. Reported that the task was a bit simple and that they finished it in under 30 minutes even though not knowing the code.

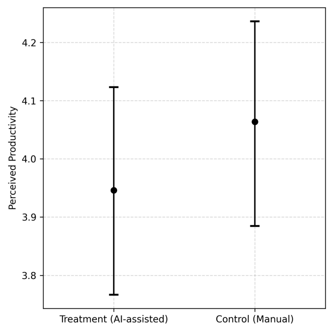
anon146 Evolved **!AI-dev** and finished the task in 88 minutes. Changed 4 files in a single commit.

anon144 Evolved **!AI-dev** and finished the task in 3 hours. Changed 12 files in 21 commits.

> *"In total I think I spent about 2.5 hours on the actual task [...] I had some initial set up issues the previous day and couldn't get the project to build correctly. I probably spent 45 mins in total messing mainly with IntelliJ config."* — `anon144, Task 2, Treatment, Pro+Advanced, R5, 290 LoC, (4.5h, CH=8.61, TC=69%, PP=5.0)`



(a) Distribution of Perceived Productivity (PP).



(b) 95% confidence intervals.

Fig. 15: Task 2 Perceived Productivity.

Frequentist Analysis

Figure 15b shows 95% confidence intervals for the mean PP. The treatment group had a mean of 3.95 (95% CI: 3.77–4.12), while the control group had a mean of 4.06 (95% CI: 3.88–4.24). Cohen's d was -0.21, indicating a *small negative effect size*. The difference was *not statistically significant* (Welch's t-test, p=0.37). Given our limited statistical power (see Section 3.4.1), our study cannot reliably detect differences corresponding to such small effect sizes.

Bayesian Analysis

As explained in section 3.4.2, our ordered logistic model infers a latent productivity score on a continuous scale for each developer. Figure 16 illustrates the relationship between inferred productivity and responses for question 1 (Q2-7a in Table 2), other questions follow the same model (taking the two inverted questions into account). To make our estimates a bit more intuitive, we report them on a scale of standard deviations of latent productivity.

For Task 1, using an AI assistant has a clear positive effect on latent productivity for habitual AI users, as $P(\Delta > 0) > 99\%$, with a mean posterior effect of +1.71 standard deviations in latent productivity (CrI: $[+1.44, +1.97]$). In other words, AI-assisted users felt more productive when working with their preferred tools. However, this positive effect on habitual AI users does not carry over to the manual evolution in Task 2 ($P(\Delta < 0) = 84\%$, with a wide posterior distribution. In Bayesian terms, this pattern suggests that negative effects are more likely than positive ones, but the evidence is too weak for firm conclusions. The posterior mean effect is -1.26 standard deviations (95% CrI: $[-2.46, +2.31]$) when manually evolving solutions by AI-assisted habitual AI users. Our sensitivity analysis shows similar results with optimistic, pessimistic, and neutral priors.

When looking at developer skill, we observed that developers with higher skill clearly reported feeling less productive in Task 1 ($P(\Delta < 0) = 98\%$). The mean effect was -1.24 standard deviations (CrI: $[-1.58, -0.87]$). As one might expect, how highly-skilled *Dev*1s perceived their own productivity did not have a significant effect on *Dev*2s' perceptions ($P(\Delta < 0) = 66\%$). The mean effect was -0.11 standard deviations (CrI: $[-2.82, +2.58]$).

To clarify how this influences the responses to productivity questions, we compare two simulations for habitual AI users with high Java proficiency and compare outcomes for an **AI-dev** and **!AI-dev**. Figure 17 presents a comparison of predicted responses in both cases. When evolving an AI-assisted solution, the model predicts that developers are 3 percentage points less likely to answer 5 ("fully agree") to each of the questions. Instead, the developer is slightly more likely to answer 3 ("neutral") or 4 ("agree"). For question 1, the probability of answering 5 shifts from 51% to 47%. For question 9, the probability of answering 3 shifts from 28% to 31%. The effects are small.

Quantitative Results – Task 2 Perceived Productivity (Trt-Ctrl)

- **Frequentist.** Mean difference = -0.10; 95% CI $[-0.40, +0.10]$;
  Welch's t-test $p = 0.37$; Cohen's $d = -0.21$.
- **Bayesian.** For habitual AI users in Task 1: $P(\Delta < 0) = 84\%$;
  Posterior mean change = -1.26 SD; 95% CrI: $[-2.46, +2.31]$.
- Both analyses are consistent with a null effect on PP in Task 2.

4.3 RQ2: Higher Quality Upon Evolution?

This section reports results related to the two dependent variables CodeHealth and Test Coverage (see Figure 2). We follow the same structure as for RQ1, including the use of textbox summaries described in Section 4.2.

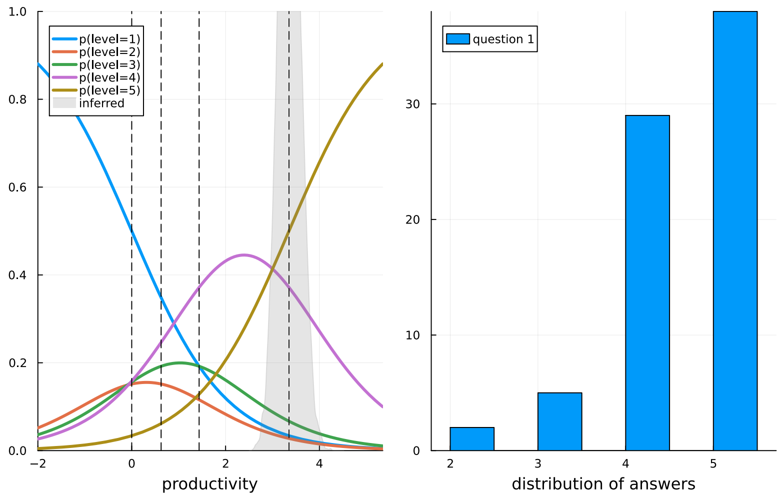


Fig. 16: Relationship between inferred productivity and responses to question 1. The left-hand plot shows the probability of each response level being observed based on the productivity on the x-axis. The shaded area shows an inferred productivity approximately between 3-4 points. The dashed lines indicate the inferred cutoffs between response levels. Looking at the curves, we can see that for a productivity in that range, the most likely answers for question 1 are level 5 (highest curve) or 4, and perhaps a few 3. This is consistent with the observed responses, showed on the right-hand plot.

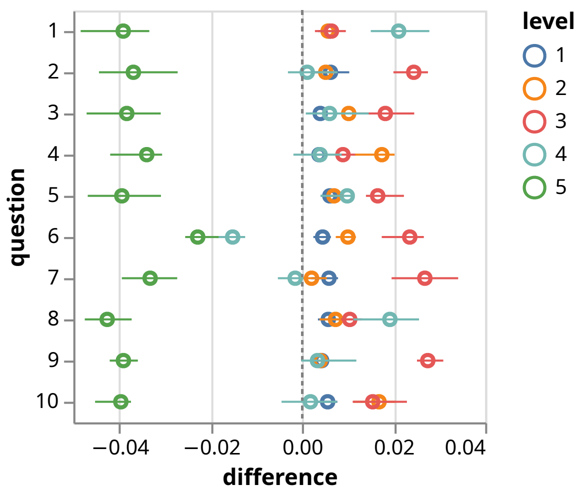
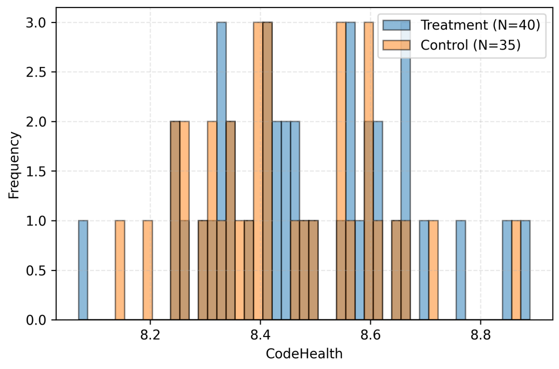


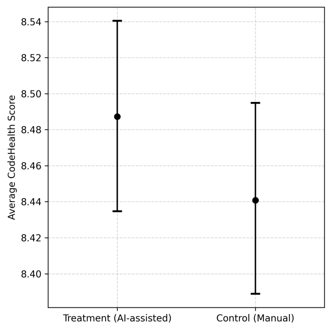
Fig. 17: Difference in predicted responses for a habitual AI user with an advanced level of Java, comparing evolution of AI-assisted vs. non-assisted solutions. Each row represents a survey question and the x-axis shows changes in response probabilities. Developers evolving AI-assisted code are about 4 percentage points less likely to answer 5 (“fully agree”) and slightly more likely to answer 3 (“neutral”).

4.3.1 CodeHealth

Figure 18a shows the distribution of CH for 75 Task 2 solutions. As described in Section 3.4.1, we consider the data approximately normally distributed. The lowest observed CH (8.07) was submitted by a participant who evolved AI-assisted code. At the other end of the spectrum, three participants submitted solutions with CH scores greater than 8.8.



(a) Distribution of CodeHealth (CH).



(b) 95% confidence intervals.

Fig. 18: Task 2 CodeHealth.

Certain patterns emerge in how CH evolves for participants in the control and treatment groups, as shown in Table 8. The treatment group built on codebases developed by **AI-devs**, while the control group worked on code from **!AI-devs**. The CH characteristics of those Task 1 solutions are detailed in Section 4.4.2 (see Tables 9 and 10).

We observe that the treatment group degraded the *Complex Method* code smell significantly more often (55%, 22 solutions) compared to the control group (31%, 11 solutions). This suggests that **AI-devs** tended to include methods with multiple logical paths, and that evolving these solutions for Task 2's requirements often led to the addition of yet another logical path. Manual inspection confirms that this additional logical path is generally related to the handling of the new `cost` parameter in the `search` method (see the Task 2 description in Appendix A.2). At the same time, the treatment group also removes more *Complex Method* code smells (20%, 8 solutions) than the control group (9%, 3 solutions), indicating that more opportunities for simplification existed in the **AI-devs** code.

The *Complex Conditional* code smell was significantly introduced more frequently in the treatment group (18%, 7 solutions) compared to the control group (3%, 1 solution). This pattern typically emerged when participants extended existing null-checks (for `query` from the base system and `time` from Task 1) with a third condition for the Task 2 `cost` parameter, resulting in a long compound conditional expression.

Alternative solutions used cleaner branching strategies, separating the `cost` and `cost` query logic.

Finally, the *Primitive Obsession* code smell appeared significantly more frequently in the control group (26%, 9 solutions) than in the treatment group (8%, 3 solutions). Manual inspection suggests that **!AI-devs** tended to extract smaller methods, leading to more parameter passing – typically simple types such as *int*, *long*, and *String* for the search time, cost, and query arguments. In contrast, **AI-devs** more often used a functional style with fewer method boundaries, reducing the number of primitives exposed between methods.

Table 8: Overview of Task 2 changes in code smells grouped by delta types and code smells, showing *mean / std / median [min-max]*.

Delta Type	Code Smell	Treatment (n=40)	Control (n=35)
Removed	Bumpy Road Ahead	0.10 / 0.30 / 0 [0-1]	0.06 / 0.24 / 0 [0-1]
	Code Duplication	0.05 / 0.32 / 0 [0-2]	0.03 / 0.17 / 0 [0-1]
	Complex Conditional	0.03 / 0.16 / 0 [0-1]	0.03 / 0.17 / 0 [0-1]
	Complex Method	0.23 / 0.48 / 0 [0-2]	0.09 / 0.28 / 0 [0-1]
	Primitive Obsession	0.03 / 0.16 / 0 [0-1]	0.00 / 0.00 / 0 [0-0]
Improved	Complex Method	0.07 / 0.27 / 0 [0-1]	0.09 / 0.28 / 0 [0-1]
Unchanged	Bumpy Road Ahead	0.05 / 0.22 / 0 [0-1]	0.09 / 0.28 / 0 [0-1]
	Code Duplication	0.10 / 0.30 / 0 [0-1]	0.17 / 0.45 / 0 [0-2]
	Complex Method	0.03 / 0.16 / 0 [0-1]	0.09 / 0.28 / 0 [0-1]
	Primitive Obsession	0.05 / 0.22 / 0 [0-1]	0.06 / 0.24 / 0 [0-1]
Degraded	Bumpy Road Ahead	0.05 / 0.22 / 0 [0-1]	0.06 / 0.24 / 0 [0-1]
	Complex Conditional	0.03 / 0.16 / 0 [0-1]	0.00 / 0.00 / 0 [0-0]
	Complex Method	0.60 / 0.59 / 1 [0-2]	0.31 / 0.47 / 0 [0-1]
Introduced	Bumpy Road Ahead	0.07 / 0.27 / 0 [0-1]	0.00 / 0.00 / 0 [0-0]
	Code Duplication	0.10 / 0.30 / 0 [0-1]	0.11 / 0.32 / 0 [0-1]
	Complex Conditional	0.17 / 0.38 / 0 [0-1]	0.03 / 0.17 / 0 [0-1]
	Complex Method	0.15 / 0.36 / 0 [0-1]	0.37 / 0.77 / 0 [0-3]
	Primitive Obsession	0.10 / 0.38 / 0 [0-2]	0.26 / 0.44 / 0 [0-1]

Frequentist Analysis

Figure 18b shows 95% confidence intervals for the mean CH. The treatment group had a mean of 8.49 (95% CI: 8.43–8.54), while the control group had a mean of 8.44 (95% CI: 8.39–8.49). Cohen's d was 0.28, indicating a *small positive effect size*. The difference was *not statistically significant* (Welch's t-test, p=0.24). As reported for PP (see Section 4.2.2), our limited statistical power means we cannot reliably detect differences corresponding to so small effect sizes.

Bayesian Analysis

For Task 1, among habitual AI users, the posterior mean difference in CH between **AI-dev** and **!AI-dev** solutions was +0.13 points ($P(\Delta > 0) = 98\%$, CrI: $[+0.02, +0.24]$). This effect did carry over to Task 2 submissions, since submissions which were initially

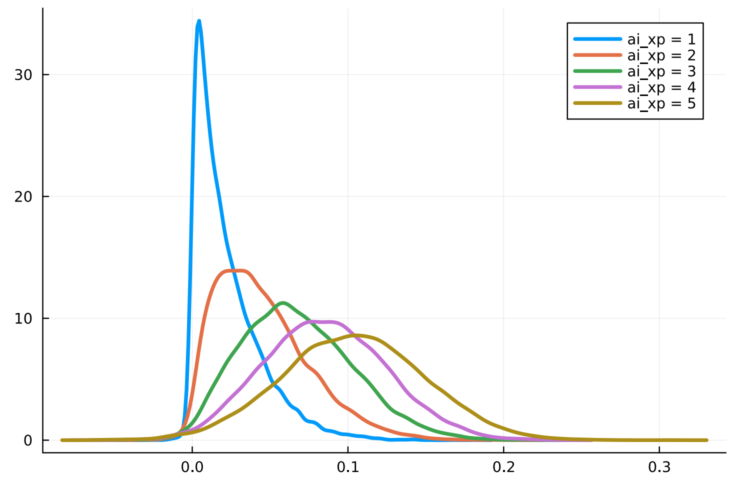


Fig. 19: Posterior effect of AI use in Task 1 on CH after manual evolution in Task 2. The effect is negligible when Task 1 developers had minimal AI experience but increases with more habitual use.

co-written by AI by a habitual AI user had a slightly higher CH score ($P(\Delta > 0) = 98\%$), with a mean effect of +0.1 (CrI: $[+0.02, +0.19]$). These estimates are small but stable under the alternative priors considered in our sensitivity analysis.

As a point of comparison, when comparing $Dev1$s with high Java proficiency versus beginners ($Dev1\_skill$ of 1 vs. 3) in Task 1, we observed that CH did not increase significantly ($P(\Delta > 0) = 69\%$). The mean effect was 0.03 points (CrI: $[-0.1, +0.13]$). An increased skill for $Dev1$ did seem to have a small positive effect on CH in Task 2 ($P(\Delta > 0) > 99\%$). The mean effect was +0.16 points (CrI: $[+0.07, +0.26]$).

Figure 19 shows the posterior distributions for the effects of using AI assistants in Task 1 on the CH of the manually evolved Task 2 solution. The five curves correspond to different levels of AI habituality (AI_xp) reported in the prescreening questionnaire. Each estimate is computed while holding Java proficiency constant, allowing a fair comparison across AI experience levels.

Quantitative Results – Task 2 CodeHealth (Trt-Ctrl)

- **Frequentist.** Mean difference = +0.05; 95% CI $[-0.03, +0.12]$; Welch's t-test $p = 0.24$; Cohen's $d = 0.28$.
- **Bayesian.** When Task 1 was completed by a habitual AI user: $P(\Delta > 0) = 0.98$; Posterior mean effect = +0.10; 95% CrI $[+0.02, +0.19]$.
- Bayesian analysis suggests a small CH advantage for the treatment group.

4.3.2 Test Coverage

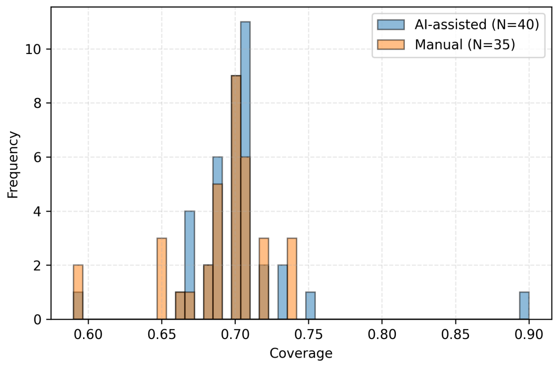
Figure 18a shows the distribution of TC for 75 Task 2 solutions. Most submissions achieved around 70% test coverage. As noted in Section 3.4.1, the data is not normally distributed. The highest observed TC was 0.90, submitted by Participant *anon050*, who evolved a Task 1 solution with TC=0.91 – the second highest score in Task 1. Note that we received no Task 2 solution building on the Task 1 solution with the highest TC (0.98). At the lower end, three participants submitted solutions with TC=0.59:

anon042 Decreased the TC of a **!AI-dev** solution by 7 percentage points. They worked on the task for 9 hours, modifying 27 files across 10 commits (703 added lines, 749 deleted lines). Although new test cases were added, the participant did not maintain TC. The average CH, however, increased by 0.09.

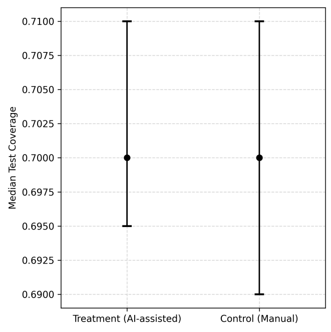
anon145 Decreased the TC of an **AI-dev** solution by 11 percentage points. They worked for 7 hours, editing 29 files in 74 commits (753 added lines, 562 deleted lines). The average CH increased by a substantial 0.60. The participant actively modified test code: *"I was refactoring test code as well, for me that is also production code, refactoring only the main service would be significantly faster."*

anon080 Increased the TC of a **!AI-dev** solution by 3 percentage points and the average CH by 0.21. They worked for 2.5 hours, and changing 21 files in 12 commits (399 added lines, 180 deleted lines).

We note that *anon042* and *anon145* were among the participants who decreased TC the most. Both were also among the participants who added the most new LoC, which might explain the phenomenon.



(a) Distribution of Test Coverage (TC).



(b) 95% confidence intervals.

Fig. 20: Task 2 Test Coverage.

Frequentist Analysis

Figure 20b shows 95% confidence intervals for the median TC. The treatment group had a median of 0.70 (95% CI: 0.695–0.71) which was matched by the control group with a median of 0.70 (95% CI: 0.69–0.74). Cliff's δ was 0.09, indicating a *negligible effect size*. The difference was *not statistically significant* (Wilcoxon rank-sum test, p=0.49).

Bayesian Analysis

For Task 1, using AI did not have a significant effect on TC ($P(\Delta > 0) = 92\%$). The posterior mean effect on the logit scale is +0.13 (CrI: $[-0.05, +0.32]$). Because we use a logit transform, these coefficients are hard to interpret directly (Gelman et al, 2021). Instead, we may look at predictions from the model: for a habitual AI user, the model predicts that the mean TC increases by 1 percentage point (from 69% to 70%) when using an AI assistant. The CrIs for predicted TC are $[66\%, 71\%]$ for **!AI-devs**, and $[68\%, 72\%]$ for **AI-devs**. Note that these estimates account for uncertainty in parameters only and do not consider residual noise.

When looking at Task 2, we also see that prior AI use has no significant effect on TC ($P(\Delta > 0) = 91\%$). The posterior mean effect is +0.10 (CrI: $[-0.04, +0.26]$). If we look at predictions from the model for *Dev*1s with high Java proficiency who are habitual AI users, the model predicts that the TC will go from 68% (CrI: $[67\%, 70\%]$) to 69% (CrI: $[68\%, 71\%]$), when using an AI assistant. We see the same results when using optimistic, neutral, and pessimistic priors.

As a point of comparison, higher Java programming skill ($Dev1\_skill$) has a significant negative effect on TC in Task 1 ($P(\Delta < 0) > 99\%$). *Dev*1s with higher skill have on average 10 percentage points less TC (CrI: $[-12pp, -5pp]$). In Task 2, this effect is slightly attenuated, while still significant ($P(\Delta < 0) = 98\%$), as the mean effect was about −4 percentage points (CrI: $[-7pp, 0pp]$).

Quantitative Results – Task 2 Test Coverage (Trt - Ctrl)

- **Frequentist.** Median difference = 0; 95% CI $[-0.01, 0.01]$;
  Wilcoxon $p = 0.49$; Cliff's $\delta = 0.09$.
- **Bayesian.** For habitual AI users in Task 1, with high Java proficiency:
  $P(\Delta > 0) = 91\%$; Model-predicted mean change = +1 pp.
- Both analyses are consistent with a null effect on TC in Task 2.

4.4 Observational Findings on AI Assistant Use in Task 1

While Task 1 is not the main target of this study, analyzing potential differences between **AI-dev** and **!AI-dev** solutions is important for interpreting the results. Furthermore, our findings can be discussed in light of the related work introduced in Section 2.3. That said, we remind the reader that the analyses in this section were not preregistered and Phase 1 was not randomized with respect to AI use, i.e., the findings are observational and do not support causal claims.

4.4.1 Task 1: Experimental Variables

Figure 21 shows the distribution of the four variables: completion time, CH, TC, and PP. Completion times (see Figure 21a) are not normally distributed and the Wilcoxon

Rank-Sum Test shows that there are significant differences ($p = 0.0029$). There is a medium effect size of using an AI-assistant (Cliff's δ=-0.42) and we observed that **AI-devs** had a 30.7% shorter median completion time. The **!AI-dev** anomaly with a very long completion time, *anon081*, explained: *"Not fluent with Java, so a lot of reading up of the basics was required."* The Bayesian analysis in Section 4.2.1 will later confirm the positive effect of using AI assistants on Task 1 completion time.

CH for **!AI-devs** is not normally distributed, but we cannot reject this hypothesis for **AI-devs**. Wilcoxon Rank-Sum Test shows that there are no significant differences between the groups ($p = 0.52$) and the effect size is negligible (Cliff's δ=0.09). However, we notice that the spread of CH is lower for **AI-devs**, which explains why the normality assumption holds. The two **AI-devs** (*anon054* and *anon126*) who reached the highest CH (9.12 and 8.88, see Figure 21b) both heavily used capable AI-assistants for more than three hours (ChatGPT+Cline+Cursor and Cursor+Tailwind AI, respectively), iterated a lot (54 and 287 test runs, respectively), and added substantial amounts of code (3,094 and 10,788 added LoC, respectively). We conclude that the AI-generated code integrated by these two participants was of high quality and raised the average CH substantially – note that the subpar 2,500 LoC of the base system is modest in comparison.

TC consistently clustered around 70% in Task 1. We note that the four highest TC scores were all obtained by **AI-devs**, while the lowest outlier corresponds to an **!AI-dev** (see Figure 21c). The highest two TC scores (98% and 91%, respectively) belong to *anon054* and *anon126* just mentioned – both users of AI assistants that go beyond code completion.

PP appears similar across the **AI-dev** and **!AI-dev** groups. However, Figure 21d shows that the six participants who reported the highest scores all worked with AI assistants. While it is reasonable that such tools increase PP – that is aligned with their purpose – we acknowledge the potential for confirmation bias. **AI-devs** may have been more inclined to evaluate their productivity positively due to their belief in the value of AI assistance.

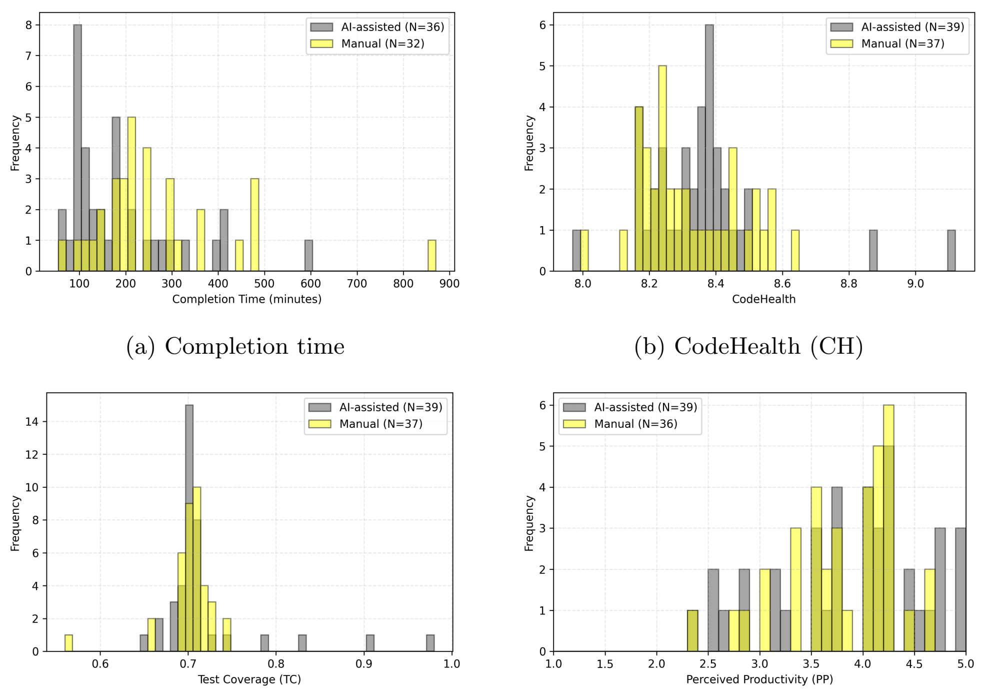
4.4.2 Task 1: Detailed Code-Level Differences

This section describes results from our detailed analysis of how **AI-devs** and **!AI-devs** changed the code in Task 1. We remind the reader that a handful of AI users generated large volumes of code, thus skewing mean values.

Regular Expressions

New *unit tests* were added by 46% of **AI-devs**, compared to 31% of **!AI-devs**. Thus, there was no clear tendency for participants to use AI assistants to supplement the existing test suite.

AI-devs added *comments* more frequently than **!AI-devs** – 79% versus 54%. AI-generated code is known for generating noisy surface-level comments. Such comments can aid the LLMs in subsequent code evolution, but offer limited value to human readers. In fact, noisy comments risk bloating the codebase and introducing discrepancies between documentation and logic if they are not kept aligned.



(a) Completion time

(b) CodeHealth (CH)

(c) Test coverage (TC)

(d) Perceived productivity (PP)

Fig. 21: Descriptive statistics of Task 1 solutions.

Finally, we observe that both **AI-devs** and **!AI-devs** tend to shift toward the *functional programming* paradigm. The proportion of solutions that introduce such constructs is 87% and 76%, respectively. Manual inspection shows that **AI-dev** submissions more frequently evolve existing usages of the Java Stream API, while **!AI-dev** submissions almost exclusively introduce new Stream APIs.

CodeScene Code Smells

Table 9 unpacks the CH of the base system and Task 1 solutions (see also Figure 21b for the CH distributions). The code smells are described in Appendix C. The "Base" column depicts that the base RecipeFinder contains: 1) one Bumpy Road, 2) eleven Complex Methods, and 3) two Code Duplication smells. As shown in the last row, the average CH is similar across the three columns. Furthermore, Table 10 presents details about how participants worked with the base system's existing code smells. Code smells can be removed, improved (modified to be less severe), unchanged (moved from one location to another), degraded (more severe), and introduced (if a new instance appears) – we refer to these five options as delta types. We observed differences between **AI-devs** and **!AI-devs**, which we discuss next.

The `search` method in Figure 23 shows the instance of the Bumpy Road – and it also triggers one of the Complex Method smells. Since Task 1 participants had to modify this method, it serves as a natural refactoring target, which we investigate in detail. We found that 72% of **AI-devs** resolved the Bumpy Road smell, compared to 46% of **!AI-devs**. Interestingly, 14% of **!AI-devs** even introduced another instance of this, whereas this never happened for **AI-devs**.

These results suggest that AI assistants effectively resolved this instance of the Bumpy Road smell while **!AI-devs** appear to struggle with that specific structural cleanup. Furthermore, 38% of the **!AI-devs** left the original code smell unchanged in the file. Manual inspection showed that this occurred when the problematic code block was merely extracted into a new method rather than being properly refactored to eliminate the underlying issue. Regarding effective refacorings, the most common resolution pattern was to rewrite the search method using a functional programming style – which was slightly more common among **AI-devs**, as discussed under the regular expressions.

On the contrary, AI assistants appear to less effectively resolve Complex Method code smells. 18% of the **AI-devs** (7 solutions) removed at least one Complex Method code smell compared to 32% of the **!AI-devs** (12 solutions). On the other hand, 54% of the **AI-devs** (21 solutions) improved smells in the Complex Method category, while 35% of the **!AI-devs** (13 solutions) did. This indicates that AI-assisted developers are likely to improve existing Complex Methods, but less able to fully resolve them.

Table 9: Unpacking Task 1 CodeHealth into code smells, showing *mean / std / median [min–max]*.

Code Smell	**Base**	**AI-dev (n=39)**	**!AI-dev (n=37)**
Bumpy Road	1	0.28 / 0.46 / 0 [0–1]	0.68 / 0.47 / 1 [0–1]
Code Duplication	2	3.79 / 5.62 / 2 [2–36]	3.08 / 3.94 / 2 [0–25]
Complex Conditionals	0	0.15 / 0.43 / 0 [0–2]	0.08 / 0.36 / 0 [0–2]
Complex Method	11	11.56 / 1.94 / 11 [6–20]	11.41 / 1.17 / 11 [6–14]
Constructor Over-Injection	0	0.03 / 0.16 / 0 [0–1]	0.00 / 0.00 / 0 [0–0]
Duplicated Assertion Blocks	0	0.03 / 0.16 / 0 [0–1]	0.00 / 0.00 / 0 [0–0]
Excess Function Args	0	0.13 / 0.66 / 0 [0–4]	0.05 / 0.33 / 0 [0–2]
Large Assertion Blocks	0	0.23 / 1.44 / 0 [0–9]	0.00 / 0.00 / 0 [0–0]
Large Method	0	0.03 / 0.16 / 0 [0–1]	0.00 / 0.00 / 0 [0–0]
Nested Complexity	0	0.08 / 0.27 / 0 [0–1]	0.00 / 0.00 / 0 [0–0]
Primitive Obsession	0	0.13 / 0.41 / 0 [0–2]	0.05 / 0.23 / 0 [0–1]
String-Heavy Args	0	0.08 / 0.27 / 0 [0–1]	0.14 / 0.42 / 0 [0–2]
Total code smells	**14**	**16.51**	**15.49**
Avg. CodeHealth	**8.3**	**8.3**	**8.4**

AI-devs show a higher tendency to introduce certain new code smells not originally present in the base RecipeFinder project. Notably, at least one *Complex Conditionals* code smell was introduced by 13% of **AI-devs** (5 solutions) compared to 5% of **!AI-devs** (2 solutions). Additionally, 8% of **AI-devs** (3 solutions) also introduced the *Nested Complexity* code smell while none of **!AI-devs** did. Furthermore, new *Code Duplication* code smells are introduced by 23% of **AI-devs** (9 solutions) while 11% of **!AI-devs** (4 solutions). Lastly, 10% of **AI-devs** (4 solutions) introduced

Table 10: Overview of Task 1 changes in code smells grouped by delta types and code smells, showing *mean / std / median [min–max]*.

Delta Type	**Code Smell**	**AI-dev (n=39)**	**!AI-dev (n=37)**
Removed	Bumpy Road	0.72 / 0.46 / 1 [0-1]	0.46 / 0.51 / 0 [0-1]
	Code Duplication	0.00 / 0.00 / 0 [0-0]	0.05 / 0.23 / 0 [0-1]
	Complex Method	0.28 / 0.86 / 0 [0-5]	0.46 / 1.04 / 0 [0-6]
Improved	Complex Method	0.54 / 0.51 / 1 [0-1]	0.41 / 0.60 / 0 [0-2]
Unchanged	Bumpy Road	0.18 / 0.39 / 0 [0-1]	0.38 / 0.49 / 0 [0-1]
	Code Duplication	0.18 / 0.39 / 0 [0-1]	0.27 / 0.45 / 0 [0-1]
	Complex Method	0.03 / 0.16 / 0 [0-1]	0.05 / 0.23 / 0 [0-1]
Degraded	Complex Method	0.31 / 0.52 / 0 [0-2]	0.43 / 0.50 / 0 [0-1]
Introduced	Bumpy Road	0.00 / 0.00 / 0 [0-0]	0.14 / 0.35 / 0 [0-1]
	Code Duplication	0.41 / 1.07 / 0 [0-6]	0.14 / 0.42 / 0 [0-2]
	Complex Conditionals	0.15 / 0.43 / 0 [0-2]	0.08 / 0.36 / 0 [0-2]
	Complex Method	0.72 / 1.52 / 0 [0-7]	0.62 / 0.83 / 0 [0-4]
	Nested Complexity	0.08 / 0.27 / 0 [0-1]	0.00 / 0.00 / 0 [0-0]
	Primitive Obsession	0.13 / 0.41 / 0 [0-2]	0.05 / 0.23 / 0 [0-1]

the *Primitive Obsession* when 5% of **!AI-devs** (2 solutions) did. Although these differences are minor–showing notable differences in mean values but high variance and identical medians–they hint that AI assistants may be prone to introducing certain code smell patterns.

Al Madi (2023) studied the readability of Copilot's generated code in a controlled experiment with students (n=21). Their results suggest that code written by a human pairing with an AI assistant is comparable in complexity and readability to code written by human pair programmers. Except for some outlier **AI-devs** who produced large volumes of code, our findings support their conclusion.

RefactoringMiner's Refactoring Operations

Table 11 presents descriptive statistics of refactoring operations identified by RefactoringMiner in the `git diffs`. The data reveal notable differences between **AI-devs** and **!AI-devs**. First, we observe that **AI-Devs** are more inclined to apply simplification refactorings, such as *Inline Variable*, *Rename Variable*, and Spring Boot annotation improvements. In contrast, **!AI-Devs** more frequently perform structural refactorings, including *Extract Class* and *Move Method* (to another class). Moreover, *Extract And Move Method*, another operation that requires deeper architectural considerations, was only recorded by **!AI-devs**. Despite these group-level patterns, we acknowledge considerable variation in how individuals refactor, no matter if they work with AI assistants or not.

PMD Linting Results

Table 12 presents an overview of PMD violations (see Appendix D) organized per set rule, i.e., related rules grouped under a common theme. By far the most frequently violated rules for both **AI-devs** and **!AI-devs** belong to the *Code Style* set, covering

Table 11: Subset of identified Task 1 refactoring operations, showing *mean / std / median [min–max]*.

Refactoring Operation	**AI-dev (n=39)**	**!AI-dev (n=37)**
Add Method Annotation	0.36 / 1.35 / 0 [0–8]	0.05 / 0.33 / 0 [0–2]
Add Param. Annotation	0.15 / 0.67 / 0 [0–4]	0.00 / 0.00 / 0 [0–0]
Change Attr. Access Modifier	0.26 / 0.79 / 0 [0–3]	0.00 / 0.00 / 0 [0–0]
Extract And Move Method	0.00 / 0.00 / 0 [0–0]	0.19 / 0.70 / 0 [0–3]
Extract Class	0.26 / 0.68 / 0 [0–3]	0.43 / 0.83 / 0 [0–4]
Extract Method	1.56 / 3.65 / 0 [0–16]	1.19 / 2.11 / 0 [0–8]
Inline Variable	0.64 / 1.74 / 0 [0–8]	0.08 / 0.36 / 0 [0–2]
Modify Param Annotation	0.33 / 1.15 / 0 [0–6]	0.00 / 0.00 / 0 [0–0]
Move Method	0.21 / 0.73 / 0 [0–4]	1.89 / 4.21 / 0 [0–19]
Rename Variable	0.87 / 1.64 / 0 [0–7]	0.30 / 0.57 / 0 [0–2]
Remove Method Modifier	0.00 / 0.00 / 0 [0–0]	0.14 / 0.48 / 0 [0–2]
Total refactorings	18.26 / 28.82 / 7 [1–126]	15.43 / 19.23 / 5 [1–69]

conventions such as naming, formatting, or method ordering. However, **AI-devs** display a much higher standard deviation (154 vs. 45) despite a lower median (258 vs. 267), confirming the impact of prolific outlier **AI-devs**. For all other rule sets, median violation counts barely differ between the groups.

Table 12 further shows a sample of particularly interesting PMD rules. We find some interesting group-level patterns. First, **AI-devs** triggered significantly fewer violations related to algorithmic complexities, such as *NPathComplexity* and *CyclomaticComplexity*. Second, **AI-devs** also produced fewer violations related to documentation rules. Third, **AI-devs**' rule violations were more concentrated in surface-level issues, such as formatting and naming. A possible explanation is that the AI-generated code tends to be more verbose, including boilerplate code that may deviate from best-practice – a pattern consistent with the *Code Style* rule set in Table 12.

Overall, it appears that contemporary use of AI assistants may reduce complexity but add stylistic clutter. This finding is in line with a study by Siddiq et al (2022), where they found that GitHub Copilot produced code with similar stylistic issues. We hypothesize that this can potentially increasing the cognitive burden of human code reviews and downstream maintainance tasks. This concludes our comparative analysis of **AI-dev** and **!AI-dev** solutions, and we are now ready to investigate Task 2's RCT results.

4.5 Analysis of Free-text Answers

The qualitative insights provided in this section enrich our data, and we triangulate them with quantitative results in the reporting. Table 5 presents the coding scheme used for the qualitative analysis of responses to Q2-8 and, in some cases, follow-up emails. In total, we coded 177 pieces of free-text input, organized into three overarching groups.

Table 12: Statistics for PMD set rules and a subset of individual rules for Task 1, showing *mean / std / median [min–max]*.

Rule	Base	AI-dev (n=39)	!AI-dev (n=37)
Best Practices:	38	49.21 / 29.83 / 40 [32–207]	42.97 / 7.65 / 40 [31–62]
UnusedAssignment	1	0.21 / 0.41 / 0 [0–1]	0.41 / 0.50 / 0 [0–1]
Code Style:	253	294.97 / 153.68 / 258 [204–1037]	279.30 / 44.54 / 267 [233–489]
AtLeastOneConstructor	11	12.62 / 6.52 / 11 [7–45]	12.08 / 2.03 / 11 [8–20]
ControlStatementBraces	23	19.67 / 3.28 / 18 [12–30]	21.73 / 3.49 / 23 [8–26]
LinguisticNaming	3	3.49 / 1.54 / 3 [3–9]	2.95 / 0.52 / 3 [0–4]
MethodArgCouldBeFinal	68	81.15 / 32.51 / 69 [53–232]	76.92 / 8.06 / 75 [69–99]
OnlyOneReturn	19	20.59 / 3.17 / 19 [15–33]	23.03 / 3.95 / 24 [12–30]
ShortVariable	25	25.79 / 4.36 / 25 [16–45]	26.03 / 2.75 / 25 [16–33]
UseExplicitTypes	0	0.08 / 0.35 / 0 [0–2]	3.35 / 12.75 / 0 [0–69]
Design:	27	26.21 / 5.84 / 25 [19–57]	26.51 / 3.43 / 27 [15–32]
AvoidCatchingGenExcep	0	0.33 / 1.46 / 0 [0–9]	0.00 / 0.00 / 0 [0–0]
CyclomaticComplexity	0	0.05 / 0.22 / 0 [0–1]	0.38 / 0.49 / 0 [0–1]
ImmutableField	4	3.05 / 1.45 / 4 [0–4]	3.54 / 1.50 / 4 [0–6]
NPathComplexity	1	0.31 / 0.47 / 0 [0–1]	0.70 / 0.46 / 1 [0–1]
Documentation:	114	122.38 / 23.56 / 115 [109–244]	124.19 / 12.68 / 118 [114–159]
CommentRequired	113	117.05 / 12.43 / 114 [102–181]	122.68 / 12.12 / 117 [113–158]
Total Rule Violations	450	512.18 / 222.17 / 456 [399–1632]	490.00 / 57.44 / 465 [438–705]

4.5.1 AI-assisted Development

Four Task 1 participants commented positively on using AI for refactoring, which lies at the core of the study. The two most positive reflections came from users of Cursor, both describing substantial improvements. For example:

> "*The AI did most of the work, really. I just had to approve its changes and ask it to solve error messages if encountered. It even provided some nice enhancement ideas when asked. I honestly felt more like a product manager than a programmer.*" — ***anon148***, `Task 1, AI-dev, Pro+Intermediate, Cursor, R4, 92 LoC, (3h, CH=8.31, TC=68%, PP=4.6)`

The other two positive participants worked with GitHub Copilot and instead described assistance at the syntax level. This contrast reflects capability differences between the first and second generations of AI assistants. Cursor encourages more chat-based programming, while GitHub Copilot rather supports traditional code completion workflows. This is reflected below:

> "*Using Copilot made the task easy. I used it to find correct syntax and refactor code to better readability.*" — ***anon103***, `Task 1, AI-dev, Pro+Intermediate, GH Copilot, R4, 129 LoC, (1.5h, CH=8.42, TC=70%, PP=4.8)`

Eight participants expressed how they found value in using AI assistants for learning and onboarding, demonstrating the relevance of this use case. Several seasoned developers had limited experience with Java and Spring Boot, and appreciated using AI assistance to get up to speed. Two of them even stated that they used AI solely during onboarding, to understand the stack and generate code examples, and then wrote all new code manually for Task 1. One explanation is that they found the onboarding challenge to be much greater than the actual coding task. A representative reflection is:

> "*I am an experienced software developer with other tech stacks, I have no professional experience with Java. This task would have been very difficult and time-consuming if not impossible for me without GitHub Copilot. It gave critical clues about where to start my debugging.*" — *anon110*, `Task 1, AI-dev [HABITUAL], Hobbyist+Intermediate, GH Copilot, R3, 134 LoC, (5h, CH=8.49, TC=70%, PP=4.4)`

However, regarding the initial debugging, we also observed contrasting comments. One participant reported that GitHub Copilot misguided them toward the wrong problem when doing initial debugging, which wasted time. A similar sentiment was shared by a student participant who highlighted that the assistant – while good at generating new code – was really bad at debugging:

> "*I tried using AI for all parts, I found that AI was good at new features, but was terrible at bug-fixing, it completely overlooked things and it took me some time to realize that the API call to retrieve the data doesn't overwrite the data, but appends it, so I would manually need to clear it after implementing the fix. AI didn't help with this at all.*" — *anon009*, `Task1, AI-dev [HABITUAL], Student+Beginner, GH Copilot+ChatGPT, R3, 46 LoC, (3h, CH=8.22, TC=70%, PP=2.5)`

Five participants mentioned how they experienced a productivity boost using AI, often in relation to onboarding into an unfamiliar codebase, as just discussed. Three of the participants, two of them using Cursor, explained how AI helped them with knowledge transfer, i.e., helping them apply their prior development experience to Java, which they had not worked with in years. For example:

> "*I have not used Java as a full-time professional language since 2011 [...] One thing where Copilot really accelerated my work was that once I understood the structure of the system I could make reference to constructs from languages and frameworks that I use on a daily basis and have it translate them into the Java and Spring [Boot] paradigm.*" — *anon133*, `Task 1, AI-dev [HABITUAL], Pro+Intermediate, GH Copilot+Grok, R4, 90 LoC, (3h, CH=8.43, TC=70%, PP=4.1)`

Obviously, not all experiences with AI assistants were positive. Ten participants mentioned limitations they found with the AI assistants during Task 1. Issues encountered include 1) *"AI getting in the way,"* 2) ChatGPT being very sensitive to prompt formulation, i.e., finding the right prompt might be slower than using web search, and 3) AI generally focusing on syntax rather than program flow. As the last statement about syntax was mentioned by *anon064*, who used GitHub Copilot, ChatGPT, JetBrains AI Assistant, and Gemini, it shows that this experience does not only hold for GitHub Copilot.

Regarding the trustworthiness of the output, two participants stressed the importance of double-checking what the AI assistants produced. For example:

> "*The interaction with the AI felt like pair programming with someone with a vast experience but lazy who sometimes gives hacky answers. I had to constantly remind the AI to use the most recent and best practice information to get better code.*" — `anon073, Task 1, AI-dev, Pro+Advanced, Cursor, R4, 2635 LoC, (4.5h, CH=8.38, TC=67%, PP=4.2)`

Finally, there was also an interesting reflection on AI-assisted development by *anon063* who was assigned to the **!AI-dev** group: *"Not relying on LLMs as I usually do these days made me reconsider their value. Honestly I'm not sure if they actually have that much value as I initially expected."* Still, we conclude that the general tone in the comments around AI assistants was moderately positive.

Key Takeaways – Task 1 Reflections on AI-assistance

- Some users of second-generation AI assistants reported success with highly automated chat-based programming.
- Several participants described very positive experiences using AI assistants for onboarding.
- Others noted limitations, including poor debugging support and prompt sensitivity – but the overall tone was moderately positive.

4.5.2 Development without AI

This section presents reflections by participants who did not use any AI assistants. The quotes include both Task 1 and Task 2 concerns.

Nine participants provided comments related to refactoring. Two expressed a sense of satisfaction or fulfilment from improving the code. This shows that developers in our study could have a positive experience regarding the tasks even without AI – which is a reassuring observation, suggesting the tasks we designed were not just a boring chore.

Two participants raised insightful comments about the difference between refactoring in this study compared to a setting in a professional development team. *anon098* explained: *"I usually don't do deeper refactorings when starting with a new codebase, respecting the current conventions and asking more experienced devs what they think about my ideas before implementing them."* Another participant went a bit further down the refactoring path, but reflected critically in hindsight:

> *"I refactored the structure of part of the code, including tests, to make it easier for my brain, but I did not apply this change to the whole codebase. This will undoubtedly create problems for whoever comes next because part of the code is in the new format and part in the old. This is a typical legacy code problem though."* — *anon027*, `Task 2, Treatment, Pro+Advanced, R4, 606 LoC, (4h, CH=8.66, TC=72%, PP=3.3)`

Seven participants mentioned how they approached the onboarding activity. As in the **AI-dev** group, several seasoned developers were either rather new to Java, or had not used it in a long time. Most comments focused on how they familiarized themselves with the tech stack, without using AI, as exemplified by the .Net and C++ developers below:

> *"I am not familiar [enough] with JPA and Java to effectively solve the coding challenge. In .NET I would have been able to finish the task in the suggested 2-4h. Most of my time was spent on discovering the framework for testing, Spring [Boot] annotations, Java standards and practice [...] I was reading one StackOverflow page after another."* — *anon012*, `Task 1, !AI-dev, Pro+Advanced, R5, 602 LoC, (3.5h, CH=8.42, TC=73%, PP=3.7)`

> *"As a C++ developer for embedded systems, Java is not my first language. I had to do some functions earlier with it, and that is how far my knowledge [goes]. [...] Figuring out the tools, environment and project structure took considerable time in the beginning. Not knowing language idioms and best practices slowed down the refactoring part and kept me from doing more."* — *anon059*, `Task 2, Control, Researcher+Beginner, R1, 161 LoC, (4h, CH=8.32, TC=66%, PP=4.0)`

Five participants working without AI assistants were somewhat put off by the limited number of test cases available. Two explicitly expressed their preferences for test-driven development. Separately, *anon116* complained about the mundane task of creating test data and felt slowed down. *anon112* considered adding more tests, but found it would require refactoring the code for testability – so they decided to finish the task without making such intrusive changes to the codebase.

Key Takeaways – Task 1 and 2 Reflections on non-AI Development

- Many participants had to ramp up on Java and Spring Boot using manual self-study, e.g., documentation and StackOverflow.
- Some participants avoided doing invasive refactoring either to respect existing conventions or due to the lack of test cases and testability.

4.5.3 Task Reflections

Beyond tool usage, many participants shared reflections on the task itself – feedback we actively elicited in Q2-8. This resulted in a total of 101 coded reflections, more than in the other two categories.

The most recurring theme was the participants' unfamiliarity with the technology stack, i.e., modern Java, Spring Boot, and Jakarta Persistence (JPA). Twenty-eight participants mentioned this in some form. In the **AI-dev** group, it was common to mention how AI assistants helped them understand the new stack, as already discussed in Section 4.5.1. Among the **!AI-devs**, it was instead common to mention which other stacks they typically used – often pointing out that the task would have been faster to solve with their familiar tools. Again, this reinforces the value of controlling for Java proficiency in the causal analysis.

Almost as common, mentioned 25 times, were reflections on the effort required to set up the environment properly before starting the task. Notably, this was only mentioned once by **AI-devs**, suggesting that AI assistants may indeed help reduce setup friction. Among the **!AI-devs**, reported issues include cloning the repository to local environments, JDK versions, IDE settings, Spring Boot and Maven, setting up the test infrastructure, GitHub Single Sign-On (SSO) over HTTPS, git configurations, and database versions. The only **AI-dev** who mentioned setup friction proposed distributing the task as a containerized solution, which was rather a suggestion for improvement. We acknowledge that containerization could have saved several participants quite some time. Nevertheless, for Task 2, randomization should mitigate any systematic impact of setup time.

Twenty-one participants mentioned that it was unclear when a solution was "good enough" for submission. The vague instructions were intentional from our side, and the random assignment should again mitigate any systematic effects. Participants reported different strategies for managing quality, including: 1) spending hours on improving large parts of the codebase, 2) delivering only the bare minimum, 3) timeboxing a fixed amount of time for refactoring, 4) prioritizing improvements to the most critical parts, 5) focusing on product code rather than test code, and 6) refactoring only areas that were touched (in line with the "Boy Scout rule" (Martin, 2009)). One participant shared an insightful comment that also captured a broader sentiment of confusion, i.e., the need for discussions with the client before undertaking larger changes:

> "*The biggest thing I was left feeling was an indecision on how far to take quality refactors/changes. If I were working on a real problem for a real client, I probably would've needed to have discussions about desirable performance characteristics, etc. that weren't possible here. That information could inform how far to take, and in what direction to take, additional changes.*" — *anon067*, `Task 2, Control, Pro+Intermediate, R3, 404 LoC, (3h, CH=8.56, TC=65%, PP=4.4)`

A related theme that recurred, often in combination with the above comments about quality expectations, was that the task was simple or even simplistic. Eighteen participants mentioned this in some way, but only four related to Task 1. It is evident that Task 2 was underwhelming to some participants:

> *"I was very confused about what I had to do during the entire experiment. Still unsure if I completed all the required tasks or not. The mentions of 3 hours and multiple sittings got me even more confused, since it probably took me less than 5 minutes for the actual code changes."* — `anon136, Task 2, Treatment, Pro+Intermediate, R4, 14 LoC, (1.5h, CH=8.42, TC=73%, PP=3.0)`

Still, note that the median Task 2 completion time was 136 minutes (see Figure 13) with a median of 56.5 added LoC for the Treatment group, and 173 minutes and 80 LoC for the Control group. The impression of Task 2 probably depended on how the preceding Task 1 was solved, as indicated by this quote mentioning a "clear prior standard":

> *"The task was extremely simple, only requiring less than 10 lines of code, and had a clear prior standard. I felt there was no room for creative thinking or really any problem solving at all."* — `anon101, Task 2, Control, Pro+Intermediate, R4, 19 LoC, (1h, CH=8.25, TC=71%, PP=3.7)`

Finally, we identified nine mentions of uncertainties in the task instructions, apart from imprecise quality expectations. Three participants mentioned details in the `README.md` file: 1) a poorly described fact that the Backend-API was externally hosted and not part of the codebase, 2) the importance of using the Spring Boot framework's @Autowire annotation, and 3) unclear instructions regarding the acceptance tests. While it is possible that some participants dropped out because of bad instructions, we consider this threat minor, as the total number of comments related to instruction problems was low.

Key Takeaways – Reflections on Tasks 1 and 2

- Many participants found the unfamiliar tech stack and setup requirements challenging, but AI assistance eased onboarding and reduced initial friction.
- Several participants were unsure how far to take refactoring and the vague quality expectations triggered a wide range of strategies.
- Although resembling previous development tasks, several participants found Task 2 simplistic – especially when Task 1 had established a high baseline.

5 Discussion

Our study is unique in its two-stage design, in which the second phase constitutes an RCT. In this phase, developers manually evolve code written by an unknown predecessor – who was either assisted by AI (treatment) or not (control). This section revisits the two RQs and discusses our findings in light of the closest related work – for which we also share findings from Task 1. Finally, we discuss the implications for research and practice.

5.1 RQ1: More Efficient Manual Evolution?

Do developers manually evolve code that has been co-developed with AI assistants more efficiently? To answer this question, we analyzed two metrics collected in the RCT and enriched the picture through free-text answers. First, the completion time for the manual evolution task (Task 2) performed by a second developer. Second, the perceived productivity of that developer, guided by the SPACE framework (Forsgren et al, 2021).

We found substantial variability in Task 2 completion time, with no significant differences between the treatment and control groups. The median completion time was 136 minutes for the Treatment group and 173 minutes for the Control group, but the confidence intervals overlapped widely and the effect size was negligible. Free-text reflections from the participants provided several explanations for the variance, including large individual variations in 1) ramp-up time needed to become familiar with the Java tech stack, 2) setup friction in configuring the local development environment, and 3) personal "definitions of done" related to code quality.

We designed the tasks around common Java technologies, but still observed a large variation in learning time, even among participants with the same self-reported Java proficiency level. Regarding setup friction, we underestimated the extent to which participants would invest in configuring their preferred local Java environment. The vague expectations on code quality were deliberate, however, and we counted on randomization to mitigate its effects – but given the sources of variation, and an underpowered RCT, it was unlikely that frequentist statistics would detect systematic differences in completion time.

Bayesian analysis tends to be a suitable alternative in such situations, providing both an estimated effect and a transparent representation of uncertainty – instead of dichotomous verdicts. The posterior mean suggests a small speedup (12.5%), but the 95% credible interval includes both zero and a small disadvantage for the treatment. In other words, at most a small and highly uncertain speed advantage emerged in our setting. Moreover, with an optimistic prior we observed a significant effect, which we interpret as a failed sensitivity analysis. We consider this another indication that the available data was insufficient to draw firm conclusions related to completion time.

As described in Section 2.3, most previous work on AI-assisted development has focused on direct productivity gains. By examining the completion times in Task 1, we can position our findings to these studies. In our study, participants using AI assistants completed Task 1 with a 30.7% shorter median completion time – a statistically significant difference corresponding to a medium effect size. This result aligns with prior studies on speedups with GitHub Copilot. For example, Peng et al (2023) reported a 55.8% speedup when implementing an HTTP server in JavaScript; Paradis et al (2025) reported a 21% speedup for C++ tasks at Google; and Chatterjee et al (2024) reported a 42.3% speedup for algorithmic Python tasks at ANZ Bank.

Furthermore, our Bayesian analysis of Task 1 supports this trend, showing that habitual AI users completed the task about 59.6% faster than the control group, matching the most optimistic results previously reported by Peng et al (2023). Thus, our study adds to the growing body of evidence that AI assistants can significantly accelerate development – providing supporting data from a different task and a different programming language.

As recommended in the SPACE framework, we complement the direct measurements with participants' self-reported perceptions of productivity. The frequentist analysis found no significant difference between treatment and control. The Bayesian analysis indicates that the most likely effect is a modest disadvantage for those evolving AI-assisted code, but with wide uncertainty. A possible explanation for this tendency is that we observed a handful of AI-assisted Task 1 participants who reported feeling very productive and also produced substantial amounts of new code. We hypothesize that Task 2 participants assigned to evolve larger solutions experienced longer onboarding periods – and thus felt less productive. While we find no clear patterns supporting this in the data, previous work highlights code size as a (or the) key maintainability challenge (Sjoberg et al, 2013). The risk that widespread adoption of AI-assistants will inflate codebases – at an unprecedented scale – is an important trend to closely monitor in the near future.

Several previous studies have examined the impact of AI assistants on perceived productivity. Ziegler et al (2024) surveyed over 17,000 GitHub Copilot users and reported a large positive impact across different dimensions. Similarly, Liang et al (2024) and Butler et al (2025) found that developers generally feel productivity gains using the same assistant. Weisz et al (2025) contrasts the picture somewhat for the watsonx Code Assistant, showing that while most developers feel faster with the tool, a large fraction feel less effective. Our Task 1 results (see Figure 21d reveal large individual variation. On the other hand, we found that AI-assisted developers felt more productive, and they are overrepresented among the participants who reported the highest scores. Free-text answers showed that some of them worked with more autonomous AI assistants, approaching the third-generation coding agents described in Section 2.2. Clearly, we need more empirical research on this emerging category of more capable tools.

RQ1: Is AI-assisted code more efficient to evolve manually?

Our RCT provides no reliable evidence that AI-assisted code differs in manual evolution efficiency. Both completion times and perceived productivity in Task 2 were consistent with a null effect, with at most small and highly uncertain differences between treatment and control.

5.2 RQ2: Higher Quality Upon Evolution?

Does code co-developed with AI assistants result in higher quality upon manual evolution? Again, we addressed this question by analyzing two metrics from the RCT, complemented by free-text answers. First, the average CodeHealth of the resulting code after the manual evolution task (Task 2). Second, the test coverage of that same Task 2 solution.

Using frequentist analysis, we found no evidence that Task 1 AI usage affected Task 2 average CodeHealth. One key source of quality variability, as discussed in Section 4.5.3, is that participants interpreted the deliberately vague task instructions differently, leading to a varied judgment about when the task was "good enough" to submit. The sample size was not large enough for randomization to fully mitigate this effect.

As with the completion time, the Bayesian analysis provides more nuanced insights. We found that the more AI-skilled the Task 1 participants were, the stronger the effect of their AI usage on Task 2 CodeHealth. For habitual AI users, the estimated mean effect was a significant absolute increase of +0.10 in CodeHealth. While the effect may appear small, we should note that this signal corresponds to half a standard deviation, detected in an average over about 50 files – and the effect is robust across priors. Still, building on code developed by a proficient Java developer mattered even more.

The findings show that developers vary in their ability to work effectively with AI assistants. While this is not surprising, it suggests that training developers to properly use these tools remains important in the current AI era. Adding to that, we also found that the Task 1 developer's Java proficiency had a stronger influence on Task 2 outcomes than AI-usage, even among habitual AI users. Human development expertise clearly still matters when working with first- and second-generation AI assistants. How this finding translates to third-generation coding agents is an open question for future work.

Previous research has reported mixed findings on how developer seniority affects the impact of AI assistants. The early work by Peng et al (2023) reported that junior developers benefited the most from GitHub Copilot, while Paradis et al (2025) reported greater speedups among senior developers at Google. Although our study includes a few juniors, we contribute a new perspective: habitual AI users using preferred tools refactor code in ways that leave lasting improvements – effects that remain even after the code is manually evolved by someone else. This provides a strong argument for equipping capable developers with refactoring-oriented AI assistants to support long-term maintainability.

Finally, we complemented the CodeHealth measurements with test coverage. We found no significant differences in Task 2 test coverage between the treatment and control groups, with both achieving a median coverage of 70%. Some AI-assisted Task 1 participants reached high coverage scores, but no free-text answers elaborated on the use of AI assistants to support testing, despite this being a realistic use case (Ouedraogo et al, 2024). However, some participants mentioned issues related to testing, such as missing test cases or poor testability. The Bayesian analysis surprisingly revealed significantly lower test coverage in Task 2 solutions preceded by highly proficient Java developers in Task 1. Given the lack of a plausible causal explanation, we consider this to be a spurious result.

RQ2: Does prior AI use improve quality after manual evolution?

Our RCT provides no reliable frequentist evidence that prior AI-assisted development affects CodeHealth after continued manual evolution. Bayesian analysis suggests a small CodeHealth improvement when the original Task 1 solution was co-developed with AI by a habitual AI user. Test coverage was essentially identical between groups.

5.3 Implications for Industry Practice

Our study has resulted in several novel insights backed by empirical data. But what does all this mean in practice? We conclude the section by discussing three interpretations for industry: one relieving finding, one risk that needs to be managed, and one challenge that remains open.

1) AI assistants tend to help with file-level maintainability issues

From a maintainability perspective, our findings suggest that developers who prefer working with AI assistants should continue doing so. We observe no systematic maintainability issues for the next developer continuing to evolve the code down the line. On the contrary, habitual AI users appear to use these supporting tools in ways that benefit future manual evolution.

We speculate that AI assistants homogenize code into standard constructs. This is supported by a recent study on LLM creativity by Haase et al (2025), which shows homogenization effects across LLMs, i.e., output from models lacks originality and clusters around similar solutions. This can be a major benefit for maintainability. Tornhill's notion of *"Beauty in Code"* (Tornhill, 2024) defined high-quality code by its lack of surprises and the ease with which a developer can form a correct mental model of its behavior. If AI assistants successfully shave off surprises, they offer a substantial maintainability advantage by reinforcing idiomatic language patterns and avoiding novel solutions to known problems. While our work highlights the value of AI assistants among capable developers, it is reasonable that these "surprise shaving" benefits would manifest at least as much among less skilled developers.

2) Reckless use of AI assistants might quickly bloat codebases

Generative AI is a new power tool in the developer's toolbox. But like any power tool, it can cause serious damage when misused. With the cost of creating new code nearing zero, the efforts to understand and maintain the code remain. This asymmetry risks bloating codebases with redundant logic, abandoned attempts, and code with questionable purpose. User discipline and project-wide retention policies will be critical in a time when we will inevitably see examples of "firehose generation" of new code.

Even if individual files become highly maintainable, the sheer code volume may undermine system-level maintainability. Code volume is a primary driver of complexity (Sjoberg et al, 2013), and AI-assisted development can accelerate growth at an unprecedented pace. One clear risk of unrestrained code synthesis is the unnoticed introduction of code clones – a long-standing maintainability concern, even if empirical studies are not conclusive on their severity (Rahman et al, 2012). Regardless, the total amount of code on the planet will reach new magnitudes, potentially triggering unforeseen phenomena as tipping points in size are crossed.

We see two primary ways to mitigate the risks. First, organizations must continuously monitor the inflow of new code into the repositories. Given the expected adoption of AI-assisted generation, tools are essential since the output volume will surpass what humans can oversee. Beyond volume, tools should also assess various quality aspects such as clone rates and security vulnerabilities. Tool vendors are already exploring how to support organizations in scaling AI adoption reliably. Second, developers should receive training to ensure disciplined AI usage. Guardrails, both technical and procedural, will help embed best practices in the workflows of the AI era. Interestingly, as several participants in our study noted, AI assistants can also be helpful for onboarding and understanding new code – suggesting that these tools may both contribute to the code size challenge and help us navigate its effects. Time will tell where the new equilibrium settles.

3) Over-reliance on AI assistants might erode skills and understanding

Neither industry nor higher education institutions yet know how to proceed with knowledge management in the AI-assisted era (Franklin et al, 2025). What skills will be central for future generations of "LLM-native" developers? How will sustained use of AI assistants affect software craftsmanship and deep understanding of complex software systems? More research along the lines of Abrahao et al (2025) is clearly needed to carefully examine what AI assistance enhances, retrieves, reverses, and obsolesces. One thing is clear: AI assistance is here to stay. The alluring simplicity of working with the tools makes the path of least resistance difficult to resist.

But as demonstrated in a recent electroencephalography experiment by Kosmyna et al (2025) (N=54), some resistance is vital for learning. In the context of essay writing, they found that participants assisted by ChatGPT showed significantly lower cognitive activity than those using web search or no tools at all. Moreover, the LLM-assisted participants underperformed in remembering details of their work. The authors refer to this as *cognitive debt*, i.e., mental effort is deferred in the short term but accumulates long-term costs, such as diminished critical inquiry and decreased creativity.

While AI-assisted development does not appear to introduce code-level technical debt, we warn about a future with build-up of cognitive debt in software organizations. The repercussions could be massive if developers end up with only a shallow understanding of codebases increasingly generated by machines. Over time, this might not only threaten maintainability but also impede innovation. To counter this risk, we should revisit decades-old research on "innovative software engineering environments" (Ambriola et al, 1991) and explore how to best support human inventiveness with the combinatorial creativity that LLMs excel at.

Finally, we recommend that organizations adopt a strategic approach to knowledge management in this new era. Tool support will be needed for provenance tracking, e.g., to monitor where only little human effort has been involved. Teams must make deliberate decisions about which components should remain under human cognitive control. Mitigating developer deskilling will become a priority when convenience drives LLM usage – "keepskilling" may emerge as a complement to upskilling and reskilling. There is a new dawn for strategic HR in software organizations, where policies ensure that developer maintain their core skills. This is analogous to pilots who are encouraged to manually land aircraft from time to time to retain their flying skills. We find this to be a substantial challenge for the future of software engineering.

6 Limitations and Threats to Validity

Selecting an empirical research method for a software engineering study always involves trade-offs. By opting for a controlled experiment, we prioritize control over realism. That said, we believe our Java development tasks are realistic and span development durations. The participants also largely agreed that they resembled prior development tasks (see Figure 12).

While we argue that our tasks are realistic, they represent only a slice of real-world Java development. Before discussing specific threats, we acknowledge four important limitations. First, our two-phase study only allows us to observe short-term maintainability effects. Maintainability is obviously part of the long game, but investigating long-term consequences requires longitudinal case studies. Second, our study focuses on individual developers. Although software engineering is inherently social, team dynamics falls outside our scope. Still, our design includes a rare controlled hand-off between developers. Third, the RecipeFinder system was kept small to allow reasonable onboarding times. Studying architectural considerations requires larger systems that deserve case study research. Fourth, we did not investigate security in this study, although previous research found that developers who work with AI assistants tend to write less secure code (Perry et al, 2023).

We organize the remaining discussion according to the categories proposed by the ACM SIGSOFT Empirical Standards (ACM SIGSOFT, 2024b).

6.1 Construct Validity

A backbone construct in our study is *AI-assisted development*. AI assistants vary in interaction mode and capability, and Task 1 participants might have interpreted the concept differently. We mitigated this by providing concrete tool examples in the Task 1 instructions and again in the exit questionnaire (Q2-3). We also asked participants to report how frequently they used their assistant (Q2-4). Moreover, the main target of this study is Task 2, in which participants were randomly assigned to evolve solutions that had been AI-assisted in a variety of ways. We consider this threat minor.

Section 2.1 explains how we define and measure the complex constructs of maintainability and productivity. We build on state-of-the-art research and operationalize the constructs using a combination of completion time, objective CodeHealth, and subjective perceived productivity guided by the SPACE framework. We consider both constructs valid, and we argue that the completeness of our measurement approach is sufficient for the scope of this study.

Still, we highlight two threats related to completion time and perceived productivity. First, as discussed in Section 3.3.2, many participants did not complete the task in one uninterrupted session (Q2-1). To remedy this protocol violation, we asked them to provide their best time estimates. Since these participants tended to have longer completion times, we believe that they generally overestimated the time it took to complete the task. However, this risk is mitigated: in Task 1 the distribution of these time estimates (instead of git-based time stamps) was evenly distributed (19 **AI-devs** and 20 **!AI-devs**), and in Task 2, randomization should balance the bias, and we directly controlled for the effect in the Bayesian analysis. Second, we speculate that **AI-devs** may have been more inclined to evaluate their productivity positively due to personal beliefs in the value of AI assistance. We accept this second threat.

Finally, our qualitative analysis of free-text answers introduces construct threats. Coding activities involve interpretations and may thus reflect the researchers' expectations. To mitigate this threat, we employed two coders and iteratively developed the coding scheme. For the qualitative analysis of source code solutions, five of the authors were involved.

6.2 Internal Validity

As we conducted a controlled experiment, the internal validity is our main concern. RCTs are textbook examples of designs that maximize internal validity, although Bayesian causal analysis is equally potent. The core argument for high internal validity lies in the random assignment in Task 2, which guards against confounding threats and balances known and unknown factors between the treatment and control groups. Still, we elaborate on the most important threats below.

Most importantly, we identified substantial variation in participants' implicit definitions of "good enough" code quality for submission. As discussed in Section 4.5.3, participants differed in how much effort they chose to invest in both Task 1 and Task 2. The vague quality requirement was a deliberate design choice to provide realism, and we wanted participants to deliver according to their own quality bars. While randomization should mitigate these effects, we acknowledge that interpretations ranged from treating the assignment "as if I were hired for a job" to providing the bare minimum effort. In the worst case, randomization may have failed to distribute these differences evenly between the two experimental groups.

Participants completed tasks remotely, and we cannot guarantee protocol adherence. Regarding AI-assistance, Q2-2 caught one violation, i.e., an **!AI-dev** had to be moved to the **AI-dev** group. While we detected no indications of other violations, e.g., use of AI assistants in Task 2, we cannot entirely rule them out. Likewise, some participants may have received other types of assistance, such as pairing up with senior friends to solve the tasks collaboratively.

As reported in Section 3.3.2, one **AI-dev** (*anon126*) submitted two Task 1 solutions. The two solutions are very different in size and completion time, with the second being minimal in comparison. Thus, we accept to treat these as two separate inputs to Task 2. This design decision affects Task 1 analysis as there is obviously a learning effect at play that we choose to accept.

Given the home-based experiment setting, we expected some unusual variation points. Indeed, the free-text answers reveal some anomalies that surely had an impact on the outcome of the tasks. Self-reported anomalies include family business, such as taking care of babies, drinking a glass of wine while completing the task, and having the TV on in the background. Again, we trust that randomization can properly control these effects.

6.3 External Validity

The results from our study cannot be generalized to all software development contexts. Nonetheless, we argue that our tasks reflect a realistic scenario: progressing code previously developed by an unknown contributor. This is a common assignment in industry. Moreover, since 92% of our participants are professional developers, and most of them between 30 and 49 years of age, we avoid the common debate about the validity of using students as subjects in controlled experiments (Feldt et al, 2018).

Still, the main threats to external validity relate to our system and task design. RecipeFinder is a small Java web application built on Spring Boot. We believe this codebase is the largest feasible size for participants to understand and extend within a reasonable time. However, the value of AI assistants may change with system size. In principle, AI assistants may be *more valuable* as information navigation tools when a maintainer needs to onboard a vast codebase to add a new feature. On the other hand, very large codebases do not fit the context window of an LLM, so AI assistants only see partial views of the architecture and a small subset of files at a time. As a result, these tools may be *less capable* at supporting cross-cutting, architecture-level maintenance in industrial-scale systems than in RecipeFinder.

In addition, our study does not provide any insight into what would happen if also the Task 2 developers used AI assistants. This might be standard practice in the near future and should be the focus of future work. Another consideration is our choice of a tech stack. The results might differ in other languages or problem domains. For example, our results might not generalize to algorithmic challenges in C++ or Python. We note that many evaluations of LLM programming capabilities have targeted competitive programming, e.g., AlphaCode (Li et al, 2022), but we are instead happy to provide results for tasks that we co-designed with senior developers to ensure practically relevant research (Garousi et al, 2020).

Another threat to the external validity concerns the specific AI assistants used. We did not restrict participants in the **AI-dev** group to any particular assistants, as we prioritized allowing them to use their preferred tools. Table 6 shows that a variety of tools were used, building on different underlying LLMs. Given this distribution, our findings primarily reflect the impact of mature commercial assistants. We cannot claim that the same results would hold for smaller, cheaper, or less sophisticated models running locally. Such models may generate lower-quality code that could potentially challenge downstream maintainability.

Finally, we acknowledge sampling bias in the demographics of the participants. Our sample is heavily male (95%), a stronger imbalance than in the general developer population. It is possible that societal roles make women less likely to participate in voluntary experiments of this nature. Geographic representation is also uneven. Large countries with substantial developer workforces are missing entirely, such as China, Japan, South Korea, and Iran. Our sample reflects the reach of the last author's English-language YouTube channel. Future work should broaden demographic representation.

6.4 Conclusion Validity

We used mixed-method research, combining objective and subjective measures, to assess the maintainability effects of AI-assisted development. We applied randomization, data cleaning, robust statistical methods, and multi-researcher coding for qualitative analysis.

A central limitation concerns the statistical power of our RCT. As quantified in Section 3.4.1, our realized sample size of 75 Phase 2 participants yields 80% power only for relatively large standardized mean differences. Hence, the non-significant results in Section 4 rule out large effects of prior AI assistance on Task 2 outcomes, but remain inconclusive with respect to small or moderate effects. We therefore interpret our frequentist findings as evidence against large effects, rather than as evidence for the absence of any effect.

Beyond this, some additional risks remain. First, a fundamental assumption of statistical analysis is that observations are independent. We assume that the participants completed their tasks without communicating during or after the assignment. This cannot be verified, but the wide geographical spread and the variety of email domains suggest that the risk is minor. There is a group of participants from Equal Experts, but we trust that they did not discuss the tasks.

Second, several of our preregistered statistical tests assume normality of the underlying data. After running Shapiro-Wilk tests (see Section 3.4.1), we found that the completion time and the test coverage violated this assumption. In these cases, we used non-parametric alternatives in the frequentist analysis. While most previous work reports mean completion times, which can be misleading for skewed distributions, we discuss medians. Moreover, our parallel Bayesian analysis relaxes the distributional assumptions and supports the robustness of our findings.

Third, our analysis of perceived productivity is based on a Likert scale. We verified high internal consistency (Cronbach's $\alpha > 0.85$), but calculating means from ordinal data remains contested. We mitigated this threat by method triangulation, as the Bayesian analysis modeled a latent productivity variable based on the individual Likert items.

Finally, the collection of free-text responses adds richness to our results but might introduce self-selection bias. Participants who were more motivated, or opinionated, probably submitted more detailed reflections. We consider this threat acceptable, as we use these responses to enrich our understanding, rather than to support any statistical inference.

7 Conclusion and Future Work

We set out to investigate the impact of AI assistants on software maintainability. To this end, we formulated two research questions targeting evolution efficiency and code quality, respectively. We conducted a preregistered two-phase controlled experiment. In Phase 2, a randomized control trial (N=75), participants manually evolved a Phase 1 solution developed by someone else – who had either worked with AI assistants such as GitHub Copilot, ChatGPT, and Cursor (treatment), or not (control). In total, we collected 151 solutions to our realistic development tasks, 95.4% of which were completed by professional software developers.

Our results provide no clear evidence that code co-developed with AI assistants is more efficient to evolve manually. Task 2 completion times were highly variable – largely due to variations in learning and setup times – and any speed advantage for evolving AI-assisted code was small and statistically unreliable. For code quality, traditional frequentist analyses found no significant differences between treatment and control. Bayesian analysis, however, suggests a small CodeHealth improvement when the original Task 1 solution was co-developed with AI by a habitual AI user. These findings indicate that, in our setting, prior AI use neither clearly improves nor degrades downstream manual evolution, with at most a small positive signal for CodeHealth in this specific subgroup.

We found additional evidence that individual variation in AI proficiency matters: in Phase 1, the posterior mean effect on completion time for habitual AI users was a 55.9% speedup, compared to a 30.7% median decrease across the entire sample. We conclude that learning to work effectively with AI assistants is a valuable developer skill – and that such proficiency may even benefit others working manually downstream. But being a proficient Java developer mattered even more in our study.

Some Phase 1 participants assisted by chat-based programming shared enthusiastic comments about their experience. Developers who reported the highest levels of perceived productivity all worked with AI assistants that went beyond code completion, most notably Cursor. This boost to perceived productivity did not carry over to Phase 2, where our results are consistent with a null effect. The Bayesian analysis shows a small and uncertain tendency toward lower perceived productivity when building on work by habitual AI users, but the evidence is too weak for firm conclusions. Future work should explore in what situations overly enthusiastic AI use might lead to downstream maintainability backlashes. We found no treatment effect on test coverage in Phase 2, but we observed that the few outstanding test suites in Phase 1 were all submitted by AI-assisted participants.

Our observational findings from Phase 1 adds to the growing body of evidence that AI assistants can effectively accelerate development. Moreover, we offer a cautiously reassuring signal for file-level maintainability. This is an encouraging finding, as we are convinced that AI assistants are here to stay. Within the scope of our tasks and measures, we found no indication that AI-assisted development harms maintainability at the code construct level. On the contrary, the large language model output is likely to shave off surprising or idiosyncratic code constructs that impede maintainability. The Bayesian analysis suggests that a positive maintainability effect is more probable than not, but any such effect would be at most small and highly uncertain. However, a greater concern is that reckless use of AI can bloat codebases. And large volumes of code, regardless of quality, are known to be hard for human developers to maintain.

Our concerns go beyond code volume. We also worry about the cognitive debt that over-reliance on AI assistants might introduce. If we increasingly generate code with minimal cognitive effort, developers may end up with only a shallow understanding of large, AI-generated codebases. Over time, this could erode core programming skills and probably stifle innovation. Whenever we automate an activity, something human is inevitably lost. How software organizations should navigate these neural tides in relation to knowledge management deserves substantial research attention in the next decade. And to what extent can the AI assistants help us cope with all the code they generate? This is not only an essential question for tool providers, but also for engineering managers, educators, and developers themselves.

These concerns point to several important directions for future work. The AI disruption is unfolding rapidly, and current AI assistants are already far more capable than the generation we studied half a year ago. With coding agents entering the scene, human bottlenecks will increasingly be sidelined. What will be the direct and indirect effects of this shift? Longitudinal case studies are needed to track these changes as they unfold, ideally complemented by action research that explores possible interventions in real time.

We also see a potential need for further controlled studies. We encourage replication of our work to explore several new angles. First, we did not study evolution by developers who were themselves AI-assisted. This is a very relevant scenario going forward. Second, future experiments should evaluate third-generation AI assistants, such as Claude Code or Sonargraph's AMP, which radically change developer workflows. Third, our analysis should be extended to consider security implications. This will be particularly important for the agentic era, since agent ensembles will emerge. Combining agentic workforces, e.g., through model context protocols, opens up cans of new security worms where autonomous agents may unintentionally expose vulnerabilities or be exploited by antagonistic actors. However, such security research is an endeavor quite different from our focus on maintainability.

Acknowledgment

We thank all participants of the study. Their willingness to voluntarily invest time and effort in completing the tasks was essential to this research.

This work was partly funded by the NextG2Com Competence Centre – Next-Generation Communication and Computing Infrastructures and Applications – under the Vinnova grant number 2023-00541. Nadim Hagatulah was supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation. Emma Söderberg was supported by the Swedish strategic research environment ELLIIT and the Swedish Foundation for Strategic Research (grant nbr. FFL18-0231).

Conflict of Interests

Markus Borg is employed by CodeScene, the company behind the CodeScene analysis tool used in this study, including the CodeHealth metric. Dave Hewett, Donald Graham, and Uttam Kini are consultants at Equal Experts, a company offering AI-accelerated software delivery services among other offerings. Dave Farley is an independent consultant and author who runs the YouTube channel Modern Software Engineering. The authors declare that no commercial interests compromised the scientific rigor of the study design, data collection, or analysis.

Data Availability Statement

All materials required to replicate this study, e.g., task instructions, cleaned and anonymized participant data, and the causal graph, are available on Zenodo (Equal Experts et al, 2025) under the Creative Commons Attribution 4.0 International license. A GitHub repository containing the full code base, scripts for data cleaning and analysis, and notebooks to conveniently reproduce the figures is available at https://github.com/codescene-research/echoes-of-ai-emse-2025. The contents of the GitHub repository are versioned and mirrored in the Zenodo archive. CodeScene was used to calculate the Code Health metric as an indicator of maintainability. While CodeScene is a commercial tool, academic researchers can request a free license for research purposes.

Compliance with Ethical Standards

The design of our study adheres to the essential attributes of the ACM SIGSOFT Empirical Standard "Ethics (Studies with Human Participants)" (ACM SIGSOFT, 2024a). Additionally, the peer review of the registered report (Borg et al, 2024b) acted as an independent assessment that the study design meets the ethical standards of the empirical software engineering community.

This study involved voluntary participation by developers who completed programming tasks and questionnaires. Before assigning tasks, we ensured that participants understood their privacy was protected, that the study was intended for academic publication, and that they could withdraw at any time without consequence. Several participants chose to do so. All participants gave informed consent prior to participation.

There were no anticipated risks of harm to participants. The task required an estimated time investment of 2–4 hours, similar in nature and duration to programming assessments used in recruitment. All participants who completed that task received a signed copy of Dave Farley's book "Modern Software Engineering: Doing What Works to Build Better Software Faster" (Farley, 2022) as an incentive to complete the task.

References

Abrahao S, Grundy J, Pezze M, et al (2025) Software Engineering by and for Humans in an AI Era. ACM Trans Softw Eng Methodol 34(5):129:1–129:46. https://doi.org/10.1145/3715111

ACM SIGSOFT (2024a) Ethics (Studies with Human Participants). URL https://github.com/acmsigsoft/EmpiricalStandards/blob/master/docs/supplements/EthicsHumanParticipants.md, commit f82836c

ACM SIGSOFT (2024b) Experiments (with Human Participants). URL https://github.com/acmsigsoft/EmpiricalStandards/blob/master/docs/standards/Experiments.md, commit c4dbe93

Al Madi N (2023) How Readable is Model-generated Code? Examining Readability and Visual Inspection of GitHub Copilot. In: Proc. of the 37th International Conference on Automated Software Engineering, pp 1–5, https://doi.org/10.1145/3551349.3560438

Ambriola V, Ciancarini P, Corradini A, et al (1991) Towards Innovative Software Engineering Environments. Journal of Systems and Software 14(1):17–29. https://doi.org/10.1016/0164-1212(91)90085-K

Ani ZC, Hamid ZA, Zhamri NN (2024) The Recent Trends of Research on GitHub Copilot: A Systematic Review. In: Zakaria NH, Mansor NS, Husni H, et al (eds) Computing and Informatics. Springer Nature, Singapore, pp 355–366, https://doi.org/10.1007/978-981-99-9589-9_27

Anthropic (2024) Model Context Protocol: Official Specification. Tech. rep., URL https://modelcontextprotocol.io

Avgeriou P, Kruchten P, Ozkaya I, et al (2016) Managing Technical Debt in Software Engineering (Dagstuhl Seminar 16162). Dagstuhl Reports 6(4):110–138. https://doi.org/10.4230/DagRep.6.4.110

Barke S, James MB, Polikarpova N (2023) Grounded Copilot: How Programmers Interact with Code-Generating Models. In: Proc. of the ACM on Prgramming Languages, pp 85–111, https://doi.org/10.1145/3586030

Basili V, Caldiera G, Rombach D (1994) The Goal Question Metric Approach. In: Encyclopedia of Software Engineering. p 528–532

Borg M, Tornhill A, Mones E (2023) U Owns the Code That Changes and How Marginal Owners Resolve Issues Slower in Low-Quality Source Code. In: Proc. of the 27th International Conference on Evaluation and Assessment in Software Engineering, pp 368–377, https://doi.org/10.1145/3593434.3593480

Borg M, Ezzouhri M, Tornhill A (2024a) Ghost Echoes Revealed: Benchmarking Maintainability Metrics and Machine Learning Predictions Against Human Assessments. In: Proc. of the 40th International Conference on Software Maintenance and Evolution, pp 678–688

Borg M, Hewett D, Graham D, et al (2024b) Does Co-Development with AI Assistants Lead to More Maintainable Code? A Registered Report. https://doi.org/10.48550/arXiv.2408.10758

Borg M, Pruvost I, Mones E, et al (2024c) Increasing, Not Diminishing: Investigating the Returns of Highly Maintainable Code. In: Proc. of the 7th International Conference on Technical Debt, pp 21–30

Butler J, Suh J, Haniyur S, et al (2025) Dear Diary: A Randomized Controlled Trial of Generative AI Coding Tools in the Workplace. In: Proc. of the International Conference on Software Engineering. arXiv

Bürkner PC, Charpentier E (2020) Modelling Monotonic Effects of Ordinal Predictors in Bayesian Regression Models. The British Journal of Mathematical and Statistical Psychology 73(3):420–451. https://doi.org/10.1111/bmsp.12195

Chatterjee S, Liu CL, Rowland G, et al (2024) The Impact of AI Tool on Engineering at ANZ Bank An Empirical Study on GitHub Copilot within Corporate Environment. In: Proc. of the 10th International Conference on Software Engineering (SEC), https://doi.org/10.48550/arXiv.2402.05636

Cook S, Ji H, Harrison R (2003) Software Evolution and Software Evolvability. In: Proc. of the Workshop on Software Analysis and Maintenance

Cui ZK, Demirer M, Jaffe S, et al (2025) The Effects of Generative AI on High Skilled Work: Evidence from Three Field Experiments with Software Developers. https://doi.org/10.2139/ssrn.4945566

Equal Experts, Borg M, Couderc N (2025) Does Co-Development with AI Assistants Lead to More Maintainable Code? Replication Package. https://doi.org/10.5281/zenodo.17977295

Farley D (2022) Modern Software Engineering: Doing What Works to Build Better Software Faster. Addison Wesley, Boston, MA, USA

Feldt R, Zimmermann T, Bergersen GR, et al (2018) Four Commentaries on the Use of Students and Professionals in Empirical Software Engineering Experiments. Empirical Software Engineering 23(6):3801–3820. https://doi.org/10.1007/s10664-018-9655-0

Fenton N (1994) Software Measurement: A Necessary Scientific Basis. IEEE Transactions on Software Engineering 20(3):199–206

Forsgren N, Storey MA, Maddila C, et al (2021) The SPACE of Developer Productivity: There's more to it than you think. Queue 19(1):10:20–10:48. https://doi.org/10.1145/3454122.3454124

Franklin D, Denny P, Gonzalez-Maldonado DA, et al (2025) Generative AI in Computer Science Education: Challenges and Opportunities. Cambridge University Press, Cambridge, MA, USA

Garousi V, Borg M, Oivo M (2020) Practical Relevance of Software Engineering Research: Synthesizing the Community's Voice. Empirical Software Engineering 25(3):1687–1754. https://doi.org/10.1007/s10664-020-09803-0

Gelman A, Hill J, Vehtari A (2021) Regression and Other Stories. Cambridge

Gorla D, Kumar S, Lorenzini PNR, et al (2025) CUBETESTERAI: Automated JUnit Test Generation using the LLaMA Model. In: Proc. of the 18th International Conference on Software Testing, Verification and Validation

Haase J, Hanel PHP, Pokutta S (2025) Has the Creativity of Large-Language Models Peaked? An Analysis of inter- and Intra-LLM Variability. https://doi.org/10.48550/arXiv.2504.12320, URL http://arxiv.org/abs/2504.12320

Husein RA, Aburajouh H, Catal C (2025) Large language models for code completion: A systematic literature review. Computer Standards & Interfaces 92:103917. https://doi.org/10.1016/j.csi.2024.103917

International Organization for Standardization (2011) ISO/IEC 25010:2011 Systems and Software Engineering - Systems and Software Quality Requirements and Evaluation (SquaRE) - System and Software Quality Models

Jaspan C, Sadowski C (2019) No Single Metric Captures Productivity. In: Sadowski C, Zimmermann T (eds) Rethinking Productivity in Software Engineering. Apress, Berkeley, CA, p 13–20

JetBrains (2024) The State of Developer Ecosystem 2024. Tech. rep., JetBrains s.r.o., URL https://www.jetbrains.com/lp/devecosystem-2024/

Kim G, Yegge S (2025) Vibe Coding: Building Production-Grade Software With GenAI, Chat, Agents, and Beyond. IT Revolution, to appear

Kosmyna N, Hauptmann E, Yuan YT, et al (2025) Your Brain on ChatGPT: Accumulation of Cognitive Debt when Using an AI Assistant for Essay Writing Task. https://doi.org/10.48550/arXiv.2506.08872

Lacerda G, Petrillo F, Pimenta M, et al (2020) Code Smells and Refactoring: A Tertiary Systematic Review of Challenges and Observations. Journal of Systems and Software 167:110610

Li Y, Choi D, Chung J, et al (2022) Competition-Level Code Generation With AlphaCode. Science 378(6624):1092–1097. https://doi.org/10.1126/science.abq1158

Liang JT, Yang C, Myers BA (2024) A Large-Scale Survey on the Usability of AI Programming Assistants: Successes and Challenges. In: Proc. of the IEEE/ACM 46th International Conference on Software Engineering, pp 1–13, https://doi.org/10.1145/3597503.3608128

Mantyla MV, Lassenius C (2006) Subjective Evaluation of Software Evolvability Using Code Smells: An Empirical Study. Empirical Software Engineering 11(3):395–431. https://doi.org/10.1007/s10664-006-9002-8

Martin R (2009) Clean Code: A Handbook of Agile Software Craftsmanship. Upper Saddle River, NJ

McElreath R (2020) Statistical Rethinking: A Bayesian Course with Examples in R and STAN, 2nd edn. Boca Raton, FL, USA

Ouedraogo WC, Kaboré K, Tian H, et al (2024) Large-scale, Independent and Comprehensive Study of the Power of LLMs for Test Case Generation. https://doi.org/10.48550/arXiv.2407.00225

Papke LE, Wooldridge JM (1996) Econometric Methods for Fractional Response Variables With an Application to 401(k) Plan Participation Rates. Journal of Applied Econometrics 11(6):619–632

Paradis E, Grey K, Madison Q, et al (2025) How Much Does AI Impact Development Speed? An Enterprise-based Randomized Controlled Trial. In: Proc. of the 47th International Conference on Software Engineering. arXiv, https://doi.org/10.1109/ICSE-SEIP66354.2025.00060

Peng S, Kalliamvakou E, Cihon P, et al (2023) The Impact of AI on Developer Productivity: Evidence from GitHub Copilot. https://doi.org/10.48550/arXiv.2302.06590

Perry N, Srivastava M, Kumar D, et al (2023) Do Users Write More Insecure Code with AI Assistants? In: Proc. of the 2023 ACM SIGSAC Conference on Computer and Communications Security, pp 2785–2799, https://doi.org/10.1145/3576915.3623157

Prosser D (2025) Worried About AI-Generated Code? Ask AI To Review It. Forbes URL https://www.forbes.com/sites/davidprosser/2025/05/07/worried-about-ai-generated-code-ask-ai-to-review-it/

Rahman F, Bird C, Devanbu P (2012) Clones: What Is That Smell? Empirical Software Engineering 17(4):503–530. https://doi.org/10.1007/s10664-011-9195-3

Sadowski C, Zimmermann T (eds) (2019) Rethinking Productivity in Software Engineering. Apress, Berkeley, CA, URL https://link.springer.com/10.1007/978-1-4842-4221-6

Schnappinger M, Fietzke A, Pretschner A (2020) Defining a Software Maintainability Dataset: Collecting, Aggregating and Analysing Expert Evaluations of Software Maintainability. In: Proc. of the 36th International Conference on Software Maintenance and Evolution, pp 278–289

Siddiq ML, Majumder SH, Mim MR, et al (2022) An Empirical Study of Code Smells in Transformer-based Code Generation Techniques. In: Proc. of the International Working Conference on Source Code Analysis and Manipulation, pp 71–82, https://doi.org/10.1109/SCAM55253.2022.00014

Silva A, Saavedra N, Monperrus M (2024) GitBug-Java: A Reproducible Benchmark of Recent Java Bugs. In: Proc. of the 21st International Conference on Mining Software Repositories, https://doi.org/10.48550/arXiv.2402.02961

Sjoberg D, Yamashita A, Anda B, et al (2013) Quantifying the Effect of Code Smells on Maintenance Effort. IEEE Transactions on Software Engineering 39(8):1144–1156. https://doi.org/10.1109/TSE.2012.89

Storey MA, Hoda R, Milani AMP, et al (2025) Guiding Principles for Using Mixed Methods Research in Software Engineering. https://doi.org/10.48550/arXiv.2404.06011

Tornhill A (2024) Use Beauty as a Guiding Principle. In: Your Code as a Crime Scene: Use Forensic Techniques to Arrest Defects, Bottlenecks, and Bad Design in Your Programs, 2nd edn. The Pragmatic Programmers, Raleigh, NC, USA

Tornhill A, Borg M (2022) Code Red: The Business Impact of Code Quality - A Quantitative Study of 39 Proprietary Production Codebases. In: Proc. of the 5th International Conference on Technical Debt, pp 11–20

Treude C, Storey MA (2025) Generative AI and Empirical Software Engineering: A Paradigm Shift. In: Proc. of the 2nd International Conference on AI-powered Software. arXiv, https://doi.org/10.48550/arXiv.2502.08108

Tsantalis N, Ketkar A, Dig D (2022) RefactoringMiner 2.0. IEEE Transactions on Software Engineering 48(3):930–950. https://doi.org/10.1109/TSE.2020.3007722

Weber T, Brandmaier M, Schmidt A, et al (2024) Significant Productivity Gains through Programming with Large Language Models. Proc of the ACM on Hum-Comput Interact 8(EICS):256:1–256:29. https://doi.org/10.1145/3661145

Weisz JD, Kumar SV, Muller M, et al (2025) Examining the Use and Impact of an AI Code Assistant on Developer Productivity and Experience in the Enterprise. In: Proc. of the Extended Abstracts of the CHI Conference on Human Factors in Computing Systems, pp 1–13, https://doi.org/10.1145/3706599.3706670

Ziegler A, Kalliamvakou E, Li XA, et al (2022) Productivity Assessment of Neural Code Completion. In: Proc. of the 6th ACM SIGPLAN International Symposium on Machine Programming, MAPS 2022, pp 21–29, https://doi.org/10.1145/3520312.3534864

Ziegler A, Kalliamvakou E, Li XA, et al (2024) Measuring GitHub Copilot's Impact on Productivity. Communications of the ACM 67(3):54–63. https://doi.org/10.1145/3633453

A System and Tasks Under Study

Controlled experiments on software maintainability require both a realistic system and carefully designed tasks. The system must be large enough to simulate real-world project conditions, and the tasks must reflect typical software development work. At the same time, the tasks must be possible to solve within a reasonable time. This section first describes how we designed the experimental artifacts, then outlines the two maintainability tasks, and finally provides a technical overview of the system under study.

A.1 Task Development and Context

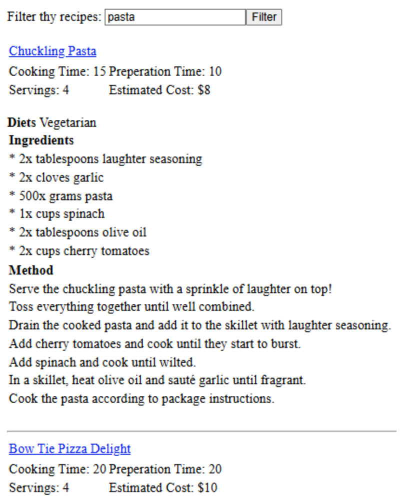
We initiated the development of the experimental artifacts with a brainstorming workshop with four senior consultants from Equal Experts and the first author. Together, we defined the design goal as specifying a typical real-world problem that should be recognizable by any professional developer. This approach helps mitigate threats to internal validity related to task familiarity. Furthermore, we specified that the system under study should exhibit the following attributes:

- code spread across multiple files;
- subpar code quality;
- unit tests present, but not complete coverage;
- API and database integration;
- involving a well-known framework;
- easily understandable domain;
- problem statement should be fun – or at least interesting;
- possible to complete in 2–4 hours – to reduce dropouts.

Guided by the above, we introduced participants to the experiment through a lightweight role-playing scenario. They took on the role of consultants hired by a hypothetical new business, Recipes4Success (R4S), whose mission is to ignite a passion for cooking among young people. Participants were told that R4S had previously engaged a software consultancy to develop a recipe service: RecipeFinder. Unfortunately, the collaboration did not meet expectations – a working web application was eventually delivered, but with poor software quality. R4S now seeks to enhance this service with new features and to establish a fruitful partnership with another consultancy, setting the stage for the participant's involvement.

RecipeFinder consists of a rudimentary web application (see Figure 22) and a supporting API. The base system offers the following features: 1) filtering the recipe list by a search term, 2) listing all recipes when the search term is blank, and 3) filtering recipes by total time (preparation time + cooking time). The service is built around a back-end API that exposes two endpoints: 1) listing all recipe IDs (with names and descriptions) and 2) getting recipe details by ID.

We stress that the task instructions and technical details presented next were refined over several iterations together with developers from Equal Experts. Senior software engineering consultants from Equal Experts co-designed and implemented the RecipeFinder system, focusing on realism: a small-scale web service with minor quality issues and only partial test coverage. Keeping the system small and scoping the maintainability tasks were necessary to ensure completion within the target time frame. We also intentionally kept the task descriptions somewhat imperfect to reflect how feature requests are often phrased in practice: R4S is not a software expert and customer assignments are rarely fully specified.



Filter thy recipes: pasta Filter

Chuckling Pasta
Cooking Time: 15 Preperation Time: 10
Servings: 4 Estimated Cost: $8

Diets Vegetarian
Ingredients
* 2x tablespoons laughter seasoning
* 2x cloves garlic
* 500x grams pasta
* 1x cups spinach
* 2x tablespoons olive oil
* 2x cups cherry tomatoes
Method
Serve the chuckling pasta with a sprinkle of laughter on top!
Toss everything together until well combined.
Drain the cooked pasta and add it to the skillet with laughter seasoning.
Add cherry tomatoes and cook until they start to burst.
Add spinach and cook until wilted.
In a skillet, heat olive oil and sauté garlic until fragrant.
Cook the pasta according to package instructions.

Bow Tie Pizza Delight
Cooking Time: 20 Preperation Time: 20
Servings: 4 Estimated Cost: $10

Fig. 22: Screenshot of the web application.

A.2 Maintainability Tasks

Participants will complete one of two tasks, both presented as high-level feature requests rather than detailed technical specifications. In Task 1, they were tasked with enhancing the existing search feature to include filtering recipes by the total time required to prepare a meal. Task 1 participants were explicitly told that changing the API was not expected. Finally, we explained to the participants that there are several design problems in the existing codebase, as well as a known bug related to the presentation of preparation and cooking times. We concluded the feature request by stating: *"We would like you to build this feature for us, and we expect a high level of quality. In particular, it should be easy to read and maintain."* We intentionally kept this quality requirement vague to reflect realistic stakeholder communication, even though this inherently leaves room for participant interpretation and an open definition of done.

Furthermore, we deliberately introduced the bug to increase realism and the flavor of subpar code quality listed as a design goal in Appendix A.1. The small defect resides in the controller code that participants must modify to implement the requested feature. Moreover, the acceptance tests for Task 1 require this defect to be fixed, and we only consider submissions that pass all tests. The injected bug thus adds a minor element of corrective maintenance and realism without biasing our comparisons, since all participants start from the same buggy baseline and only passing solutions proceed to the next phase.

In Task 2, participants were asked to further develop the Task 1 search feature by incorporating a filter for the cost per serving. Completing this task required the participant to build on the solution provided by the Task 1 participant. Note that we used the same R4S role-playing scenario for Task 2, no matter the quality of the Task 1 solution. Again, the feature request concluded with the same statement about our quality expectations. Both tasks included three acceptance tests to be executed using GitHub Actions, which target the expected behavior of the completed solution – and ensure that the Task 1 defect has been resolved.

To validate the setup, five consultants from Equal Experts pilot-tested either Task 1 or Task 2. These pilots tested the clarity of the instructions, the estimated time budget, and the supporting infrastructure (especially *snapcode.review*, described in Section 3.3.1). The exact task descriptions are available in the replication package (Equal Experts et al, 2025).

A.3 RecipeFinder Codebase: Technical Details

The base application, RecipeFinder, is a deliberately substandard Java/Spring Boot application used as the starting point for Task 1. We chose to rely on Spring Boot as it is widely regarded as a standard framework in the contemporary Java stack (Gorla et al, 2025). Features related to our tasks include Spring Data JPA for database access, Spring MVC for REST endpoints, and annotation-based dependency injection.

The total size of the base system, i.e., the codebase sent to the Task 1 participants, is 2 KLoC across 47 files. We analyzed the maintainability of the base system using CodeScene and complemented the results with low-level style checks using the PMD[19] linting tool (v7.13.0) using all Java rules[20]. We present these analyses when comparing how developers working with or without AI assistants evolved the Task 1 code in Section 4.4.2.

Figure 23 shows a Java method from `RecipeFinderController.java`, which is at the core of both tasks. In Task 1, participants modified this method to implement the new feature – and in Task 2, other participants changed it again. The code displays subpar characteristics and violations of standard Java conventions. The method contains two CodeScene code smells, i.e., Bumpy Road and Complex Method, which will be further discussed in Section 4.4.2.

The codebase contains a unit test suite that helps participants understand the behavior of key Java methods. Moreover, it provides a starting point for those who prefer test-driven development. Participants could add or modify the unit tests as they added new features. Note that the unit tests did not expose the injected defect.

Finally, to preserve the integrity of the study, i.e., preventing any leakage to AI assistants' training data (Silva et al, 2024), we chose not to host the code in a public git repository. Instead, all relevant documents and code are shared as PDF documents in the replication package (Equal Experts et al, 2025).

B Bayesian Statistical Models

This appendix provides an introduction to Bayesian analysis and presents details for our modeling of the four dependent variables.

[19] https://pmd.github.io/
[20] https://github.com/pmd/pmd/blob/e5516daad869a4895ca14e3257af15f528993bc7/pmd-core/src/main/resources/rulesets/internal/all-java.xml

Fig. 23: Original search handler in `RecipeFinderController.java` used in Task 1. The red dashes indicate the Bumpy Road code smell (bumps on lines 13, 20, and 27). The Complex Method smell is triggered due to a cyclomatic complexity of 13.

```
 1          @GetMapping("/recipes")
 2          public String search(@RequestParam(name="query", required=false) String query, Model model) {
 3  —         List<Recipe> recipes = recipeRepository.findAll();
 4  —         model.addAttribute("recipes", recipes.stream().filter(recipe -> {
 5  — —           if (query == null) {
 6  — — —             return true;
 7  — —           }
 8  — —           if (recipe.getName().toLowerCase().contains(query.toLowerCase())) return true;
 9  — —           if (recipe.getDescription().toLowerCase().contains(query.toLowerCase())) return true;
10  — —           long count = 0L;
11  — —           for (Ingredient x : recipe.getIngredients()) {
12  — — —             if (x.getName().toLowerCase().contains(query.toLowerCase())) {
13  — — — —               count++;
14  — — —             }
15  — —           }
16  — — —             if (count > 0) return true;
17  — — —             count = 0;
18  — — —             for (Method m : recipe.getMethod()) {
19  — — —             if (m.getDescription().toLowerCase().contains(query.toLowerCase())) {
20  — — — —               count++;
21  — — —             }
22  — —           }
23  — —           if (count > 0) return true;
24  — —           count = 0;
25  — —           for (Diet diet : recipe.getDiets()) {
26  — — —             if (diet.getName().toLowerCase().contains(query.toLowerCase())) {
27  — — — —               count++;
28  — — —             }
29  — —           }
30  — —           if (count > 0) return true;
31  — —           count = 0;
32  — —
33  — —           return false;
34  —         }).collect(toList());
35  —
36  —         return "recipe-search";
37          }
```

B.1 Preamble

In the models described below, we describe a generative model of the data. Each model contains a *likelihood*, which describes how we think the observed data was generated. This likelihood usually has parameters that we do not know and want to *infer*. For each of these parameters, we provide a *prior* (another distribution). Bayesian inference software (in our case, the Turing Julia library[21]) then estimates a *posterior distribution* of the likely values of the parameters. Posterior distributions are a compromise between the likelihood and the prior distributions. For example, the model below estimates the mean and variances (μ and σ) of a dataset y.

$$
\begin{aligned}
\mu &\sim \text{Normal}(0, 1) \\
\sigma &\sim \text{Exponential}(1) \\
y_i &\sim \text{Normal}(\mu, \sigma)
\end{aligned}
$$

We follow the same notation for all our models, $\alpha \sim d$ denotes that parameter α has distribution d. The last line is the likelihood, and uses parameters μ and σ, which we want to estimate, we therefore give them priors. In the following models,

[21]https://turing.ml/

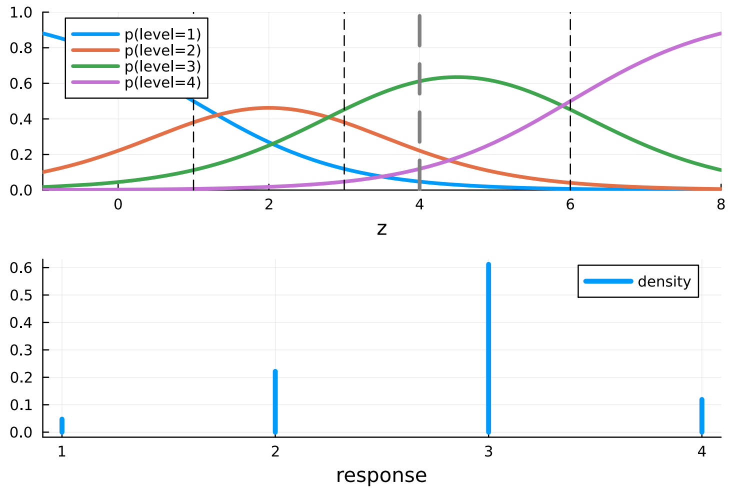


Fig. 24: Two plots, showing the relationship between the latent variable z and an ordered logistic distribution with cutoffs $[1, 3, 6]$. The top plot shows the probability of each of the 4 levels as a function of z, where the cutoffs are shown using black dashed lines. The bottom plot shows the probability of each response level for $z = 4$. The bottom plot is a slice indicated on the top plot by a gray dashed line. As z increases, higher response levels become more probable, but there is no assumption that cutoff levels are evenly separated.

we use the normal, exponential, Dirichlet, and ordered logsitic distributions. The normal distribution is the familiar "bell curve" distribution, with the difference that we denote $\text{Normal}(\mu, \sigma) = \mathcal{N}(\mu, \sigma^2)$, for simplicity. The exponential distribution covers a continuous range of positive values, we use it for variance terms, which are strictly positive.

The Dirichlet distribution is a distribution of vectors, where the sum of the vectors is equal to 1. We use it to represent monotonic effects, which we clarify with an example. Suppose we want to quantify the effect of education (BSc, MSc, or PhD) on another variable (e.g., yearly income). Modeling this with independent effects does not necessarily make sense, because people who have a master's degree usually have a bachelor's degree too. Instead, we represent the effect of education as a *cumulative sum* of effects, with two components. First, we have a parameter for the *maximum* effect of the variable (e.g., for education, the effect of having a PhD). And second, we have a vector of reals between 0 and 1 that sum to 1. This vector represents the effect of each level as a fraction of the maximum.

In our models, we usually use this distribution to represent the effect of 1) experience in Java and 2) experience in AI-assisted development. Consider the following equations:

$$\beta \sim \text{Normal}(0, 1), \;\; \boldsymbol{\lambda}_{\text{xp}} \sim \text{Dirichlet}(5, 1.0)$$

$$\theta_{\text{xp}}[j] = \left(\sum_{k=1}^{j} \lambda_{\text{xp}}[k] \right) \cdot \beta \quad \text{for } j = 1, 2, 3, 4, 5$$

$\theta_{\text{xp}}[j]$ represents the effect of having the experience level j on the outcome (not shown here). In our analyses, 1 corresponds to "Minimal experience", and 5 to "High experience". β represents the effect of the maximum level of experience in AI ("High"). $\boldsymbol{\lambda}_{\text{xp}}$ represents the "ladder" of effects. The example shows a vector of 5 values, which sum to 1, and each element represents the difference with the previous level.

The ordered logistic distribution is useful for modeling variables on a discrete scale on K levels, where the levels are ordered but we do not know how distant they are from each other. It takes two parameters: z is real number and a vector of "cutoffs" $c_1, c_2, \ldots, c_{K-1}$. It describes the probability of obtaining a discrete response level $r \in [1, 2, \ldots, K]$. In our analysis, we assume that when $z < 0$, the most probable level is the minimum, and as z increases, the probabilities of observing each of the K levels behave like displayed in Figure 24.

Next, we present the details related to the four dependent variables under study.

B.2 Completion Time

- a_i: AI usage (binary)
- $x_i \in \{1, 2, 3, 4, 5\}$: Degree of habitual AI assistant use (Q1-7a)
- $s_i \in \{1, 2, 3\}$: Developer 1 Java skill level (Q1-5)
- $u_i \in \{1, 2, 3\}$: Developer 2 interruption level (Q2-1)
- y_i: Observed completion time
- μ_β and σ_β : Mean and variance of the prior we select.

In the model below, the likelihood of the logarithm of completion time $\log(y_i)$ is modeled with a normal distribution. The mean μ_i is the sum of an intercept α, the effect of Java proficiency of the developer $\theta_{\text{skill}}[s_i]$, and the effect of their experience in AI $a_i \cdot \theta_{\text{xp}}[x_i]$ (multiplied by a_i so that this effect disappears if AI is not used ($a_i = 0$)), and the effect of interruptions $\theta_{\text{int}}[u_i]$. The notation $v[i]$ refers to vector indexing, starting at 1.

$$\alpha \sim \text{Normal}(0, 1) \tag{1}$$

$$\beta \sim \text{Normal}(\mu_\beta, \sigma_\beta), \ \boldsymbol{\lambda}_{\text{xp}} \sim \text{Dirichlet}(5, 1.0) \tag{2}$$

$$\theta_{\text{xp}}[j] = \left(\sum_{k=1}^{j} \lambda_{\text{xp}}[k] \right) \cdot \beta \quad \text{for } j = 1, 2, 3, 4, 5 \tag{3}$$

$$\gamma \sim \text{Normal}(0, 1), \ \boldsymbol{\lambda}_{\text{skill}} \sim \text{Dirichlet}(3, 1.0) \tag{4}$$

$$\theta_{\text{skill}}[j] = \left(\sum_{k=1}^{j} \lambda_{\text{skill}}[k] \right) \cdot \gamma \tag{5}$$

$$\delta \sim \text{Normal}(0, 1), \ \boldsymbol{\lambda}_{\text{int}} \sim \text{Dirichlet}(3, 1.0) \tag{6}$$

$$\theta_{\text{int}}[j] = \left(\sum_{k=1}^{j} \lambda_{\text{int}}[k]\right) \cdot \delta \tag{7}$$

$$\sigma_j \sim \text{Exponential}(1) \quad \text{for } j = 1, 2, 3 \tag{8}$$

$$\mu_i = \alpha + \theta_{\text{skill}}[s_i] + a_i \cdot \theta_{\text{xp}}[x_i] + \theta_{\text{int}}[u_i] \tag{9}$$

$$\log(y_i) \sim \text{Normal}(\mu_i, \sigma_{u_i}) \tag{10}$$

B.3 CodeHealth

- a_i: AI usage (binary)
- $x_i \in \{1, 2, 3, 4, 5\}$: Degree of habitual AI assistant use (Q1-7a)
- $s_i \in \{1, 2, 3\}$: Developer 1 Java skill level (Q1-5)
- y_i: Observed CodeHealth score
- μ_β and σ_β : Mean and variance of the prior we select.

Model:

$$\alpha \sim \text{Normal}(0, 1) \tag{11}$$

$$\beta \sim \text{Normal}(\mu_\beta, \sigma_\beta) \tag{12}$$

$$\boldsymbol{\lambda}_{\text{xp}} \sim \text{Dirichlet}(5, 1.0) \tag{13}$$

$$\theta_{\text{xp}}[j] = \left(\sum_{k=1}^{j} \lambda_{\text{xp}}[k]\right) \cdot \beta \quad \text{for } j = 1, 2, 3, 4, 5 \tag{14}$$

$$\gamma \sim \text{Normal}() \tag{15}$$

$$\boldsymbol{\lambda}_{\text{skill}} \sim \text{Dirichlet}(3, 1.0) \tag{16}$$

$$\theta_{\text{skill}}[j] = \left(\sum_{k=1}^{j} \lambda_{\text{skill}}[k]\right) \cdot \gamma \quad \text{for } j = 1, 2, 3 \tag{17}$$

$$\sigma \sim \text{Exponential}(1) \tag{18}$$

$$\mu_i = \alpha + a_i \cdot \theta_{\text{xp}}[x_i] + \theta_{\text{skill}}[s_i] \tag{19}$$

$$y_i \sim \text{Normal}(\mu_i, \sigma) \tag{20}$$

B.4 Test Coverage

- a_i: AI usage (binary)
- $x_i \in \{1, 2, 3, 4, 5\}$: Degree of habitual AI assistant use (Q1-7a)
- $s_i \in \{1, 2, 3\}$: Developer 1 Java skill level (Q1-5)
- y_i: Observed logit-transformed test coverage
- μ_β and σ_β : Mean and variance of the prior we select.

$$\alpha \sim \text{Normal}(0, 1) \tag{21}$$

$$\beta \sim \text{Normal}(\mu_\beta, \sigma_\beta) \tag{22}$$

$$\boldsymbol{\lambda}_{\text{xp}} \sim \text{Dirichlet}(5, 1.0) \tag{23}$$

$$\theta_{\text{xp}}[j] = \left(\sum_{k=1}^{j} \lambda_{\text{xp}}[k]\right) \cdot \beta \quad \text{for } j = 1,2,3,4,5 \tag{24}$$

$$\gamma \sim \text{Normal}() \tag{25}$$

$$\boldsymbol{\lambda}_{\text{skill}} \sim \text{Dirichlet}(3, 1.0) \tag{26}$$

$$\theta_{\text{skill}}[j] = \left(\sum_{k=1}^{j} \lambda_{\text{skill}}[k]\right) \cdot \gamma \quad \text{for } j = 1,2,3 \tag{27}$$

$$\sigma \sim \text{Exponential}(1) \tag{28}$$

$$\mu_i = \alpha + a_i \cdot \theta_{\text{xp}}[x_i] + \theta_{\text{skill}}[s_i] \tag{29}$$

$$\text{logit}(y_i) \sim \text{Normal}(\mu_i, \sigma) \tag{30}$$

B.5 Perceived Productivity

- a_i: AI usage (binary)
- $x_i \in \{1,2,3,4,5\}$: Degree of habitual AI assistant use (Q1-7a)
- $s_i \in \{1,2,3\}$: Developer 1 Java skill level (Q1-5)
- $y_{i,q} \in \{1,\ldots,L\}$: Ordinal response for person i on question q
- L: Number of ordinal levels
- Q: Number of questions
- μ_β and σ_β : Mean and variance of the prior we select.

$$\alpha \sim \text{Normal}(0, 1) \tag{31}$$

$$\beta \sim \text{Normal}(\mu_\beta, \sigma_\beta) \tag{32}$$

$$\boldsymbol{\lambda}_{\text{xp}} \sim \text{Dirichlet}(5, 1.0) \tag{33}$$

$$\theta_{\text{xp}}[j] = \left(\sum_{k=1}^{j} \lambda_{\text{xp}}[k]\right) \cdot \beta \quad \text{for } j = 1,2,3,4,5 \tag{34}$$

$$\gamma \sim \text{Normal}() \tag{35}$$

$$\boldsymbol{\lambda}_{\text{skill}} \sim \text{Dirichlet}(3, 1.0) \tag{36}$$

$$\theta_{\text{skill}}[j] = \left(\sum_{k=1}^{j} \lambda_{\text{skill}}[k]\right) \cdot \gamma \quad \text{for } j = 1,2,3 \tag{37}$$

$$\pi_i = \alpha + a_i \cdot \theta_{\text{xp}}[x_i] + \theta_{\text{skill}}[s_i] \tag{38}$$

$$\delta_{q,\ell} \sim \text{Exponential}(1) \quad \text{for } q = 1,\ldots,Q,\ \ell = 1,\ldots,L-2 \tag{39}$$

$$c_{q,0} = 0 \tag{40}$$

$$c_{q,\ell} = \sum_{m=1}^{\ell} \delta_{q,m} \quad \text{for } \ell = 1,\ldots,L-2 \tag{41}$$

$$y_{i,q} \sim \text{OrderedLogistic}(\pi_i,\ [c_{q,0}, c_{q,1}, \ldots, c_{q,L-2}]) \tag{42}$$

C CodeScene Code Smells

This appendix describes the CodeScene code smells observed and referenced in the article. These smells were identified using the `CodeScene CLI` tool through file-level analysis. The following commands were used: `cs check ${FILE}` for analyzing individual files, and `cs delta ${FIRST_USER_COMMIT} HEAD` for analyzing the diff of individual files between commits.

Bumpy Road (or Bumpy Road Ahead). Assigned to a function that contains multiple chunks of nested control structures. The deeper the nesting, the more bumps, and the more severe the Bumpy Road smell is. These bumps in the code represent missing abstractions and make the function harder to understand and maintain.

Complex Conditionals indicates branch expressions that combine multiple logical operations, such as conjunctions and disjunctions, within a single condition. These statements are harder to read and reason about than a well-named boolean identifier that could represent the behavior and logic of the conditions. Complex conditionals obscure the intent of the code and contribute to the overall complexity of the method in which they appear.

Complex Method is assigned to functions with high cyclomatic complexity, which means that they contain many independent logical paths. Such methods are more difficult to test thoroughly and are more prone to bugs. They often try to do too much and lack a clear separation of concerns. Splitting complex methods into smaller, more focused units improves both readability and testability.

Nested Complexity (or Deep, Nested Complexity) is a smell that arises when control structures, such as loops and conditionals, are nested within each other to multiple levels. This kind of nesting increases the cognitive effort required to understand the flow of execution.

Excessive Function Arguments is assigned when functions take too many parameters. This can indicate that the function is doing too much or that it lacks a proper abstraction to group related data. Functions with many arguments are harder to call correctly and more difficult to refactor.

Large Assertion Blocks is a test code smell that is a sign of poor structure. When many assertions are grouped together without clear separation, it can become harder to understand what each test is verifying, which reduces the effectiveness of the test suite and makes failures harder to diagnose.

Large Method indicates a method with many LoC, this typically indicates multiple responsibilities, which can make the method harder to read and should be decomposed into smaller, more cohesive units.

Code Duplication is a file-level smell that potentially makes code evolution harder. Any change must be replicated across all instances, increasing the risk of inconsistencies and bugs. Duplication also inflates the codebase, making it potentially harder to navigate and understand.

Duplicated Assertion Blocks is a test code smell similar to code duplication, making code harder to maintain. Duplicated test criteria indicate missing abstractions or a test suite that attempts to test too many things inside the same module.

Constructor Over-Injection is assigned when a constructor has many arguments. This indicates that either a unit has low cohesion or an injection of dependencies on the wrong abstraction level.

Primitive Obsession is assigned to code that uses a high degree of built-in primitives such as integers, strings, and floats. This shows a lack of domain language that encapsulates the validation and semantics of function arguments.

String-heavy Arguments is related to primitive obsession, where the heavy usage of strings could indicate a missing domain language. String is a generic type that often fails to capture the constraints of the domain object it represents.

D PMD Rules

This appendix describes the significant PMD rules violations observed and referenced in the article[22].

UnusedAssignment Triggers when a variable is assigned a value that is never used. This may indicate leftover code or incomplete logic.

AtLeastOneConstructor Flags non-static classes that do not declare an explicit constructor. Adding one makes the class definition clearer and more consistent.

ControlStatementBraces Enforces the use of curly braces in control structures such as `if` and `for` to reduce ambiguity and prevent logic errors.

LinguisticNaming Detects mismatches between identifier names and their types, such as a non-boolean variable named like a boolean. This improves naming consistency and code readability.

MethodArgCouldBeFinal

OnlyOneReturn Encourages methods to have a single return statement at the end, simplifying control flow and enhancing maintainability.

ShortVariable Flags variable names shorter than three characters unless used in conventional contexts like loop counters. Short names reduce clarity and hinder comprehension.

UseExplicitTypes Warns when the `var` keyword is used. Requiring explicit types improves code clarity and reduces cognitive overhead for readers.

AvoidCatchingGenExcep Warns against catching overly generic exceptions such as `Exception` or ‘Throwable‘, which can mask programming errors and complicate debugging.

CyclomaticComplexity Flags methods with high cyclomatic complexity (threshold: 10). Complex methods are harder to test and understand and often benefit from refactoring.

ImmutableField Identifies fields that can be marked `final` but are not. Immutability strengthens code reliability and reduces potential side effects.

NPathComplexity Calculates the number of possible execution paths through a method. A high value (threshold: 200) suggests the method may be overly complex and hard to test.

[22] https://pmd.github.io/pmd/pmd_rules_java.html

CommentRequired Ensures that key elements like public classes and methods include Javadoc comments. Proper documentation supports better understanding and long-term maintainability.