Summary

This thesis explores whether introducing friction into AI-assisted programming tools, specifically GitHub Copilot, affects users' cognitive load, confidence, frustration and task progress. Friction is defined here as intentional design elements that delay user interaction to promote deeper reflection and critical engagement with AI-generated content.

To test this, we developed CoFriction, an extension for Visual Studio Code that modifies GitHub Copilot to include three friction mechanisms:

(1) **Slow accept:** users must manually type out GitHub Copilot suggestions rather than pressing a key to accept the entire suggestion.
(2) **Inline feedback:** line-by-line AI feedback is displayed within the code editor.
(3) **Code highlighting:** AI-generated copy-pasted code is visually marked.

A between-subjects experiment was conducted with 38 Software Engineering students enrolled in a Software Master's program, split evenly between a control group using standard Copilot and a group using CoFriction. Participants completed a set of Python programming tasks within 25 minutes, after which they filled out an extended cognitive load questionnaire that also measured confidence and frustration. Afterwards they participated in semi-structured interviews to gather qualitative feedback.

Quantitative results showed no statistically significant differences between the two groups in terms of intrinsic, extraneous, or germane cognitive load, confidence, frustration, number of tasks completed and compilability of the code.

Interviews, however, revealed that participants often took note of slow accept with many stating they found it frustrating, but others also appreciated the value in a learning context. Inline feedback and code highlighting mostly went unnoticed or ignored, as well as several participants expressing dislike. Other participants, while they often did not notice the frictions, liked code highlighting's potential to separate AI code from human-written code and enjoyed the additional feedback provided through inline feedback.

Thematic analysis of interview data revealed six recurring themes, including the influence of time constraints, perceived educational value of friction, and varying trust in AI-generated code. Interestingly, many participants saw potential for friction in learning environments but considered it less useful in high-efficiency settings.

Our findings indicate that our implemented friction does not significantly affect cognitive load, frustration, confidence, and task progress in short, time-pressured settings. Qualitative responses, however, could indicate that in different settings with low time pressure, inline feedback and code highlighting could have a more noticeable effect on user cognitive engagement, while a more intrusive friction, such as slow accept is required under high time pressure. Furthermore, some participants tended to prompt first and then later try to understand the AI code if it did not compile. Some participants regretted their usage of AI, as the code became difficult to understand and had errors.

Exploring the Effect of Friction in AI Programming Tools on Cognitive Load

Mike Jensen
maltow20@student.aau.dk
Aalborg University, CS Department
Aalborg, Denmark

Rasmus Bisgaard Kristensen
rbkr20@student.aau.dk
Aalborg University, CS Department
Aalborg, Denmark

Sebastian Malthe Røn
sm20@student.aau.dk
Aalborg University, CS Department
Aalborg, Denmark

ABSTRACT

We conducted an exploratory study to investigate whether introducing friction into AI programming tools can affect cognitive load. A total of 38 Software Engineering Master's students were assigned to either a control group using standard GitHub Copilot or a friction group using a modified version with slow acceptance of code suggestions, highlighting of AI-generated code, and inline AI feedback of their code. Using programming tasks, a cognitive load questionnaire extended to measure frustration and confidence, and qualitative post-task interviews, we measured cognitive load, confidence, frustration, task progress, and qualitative reflections. Results showed no statistically significant differences between the friction and non-friction group across any metric, including intrinsic, extraneous, and germane cognitive load, confidence, frustration, or tasks started or compilation status. While participants often overlooked or ignored the more subtle friction elements, they expressed mixed views on the value of friction, some finding it disruptive, others seeing its potential in educational contexts. Our findings suggest that the types of friction we implemented do not significantly alter user cognition or behaviour in short, time-pressured settings, though qualitative responses point towards possible benefits in a different setting.

KEYWORDS

AI-assisted programming; Cognitive Load Theory; CLT; cognitive load; friction; GitHub Copilot; human-AI interaction; user reflection; programming tools; interaction design; code acceptance behaviour.

1 INTRODUCTION

Artificial Intelligence (AI) tools such as GitHub Copilot and ChatGPT are becoming increasingly embedded in software development, offering real-time code suggestions, bug fixes, and even entire implementations [2, 13, 38]. As a result, they are rapidly being adopted in professional and educational settings alike [1, 11, 27, 37]. The convenience and speed they offer are reshaping how programmers approach their tasks: from writing boilerplate code to explaining and improving code [22, 24].

However, this introduces a critical risk: programmers may begin to trust and adopt AI-generated code without sufficient review [42]. The fluid, low-friction interactions encouraged by these tools can discourage reflective thinking, potentially leading to misunderstandings, unnoticed errors, or a shallow grasp of the implemented code [34, 42]. While AI tools reduce the effort required to write code, they may also inadvertently reduce the effort users invest in understanding the code. This raises a fundamental question: can the design of AI tools encourage deeper engagement with their suggestions rather than passive acceptance?

To explore this, we conducted a study in which we introduced friction into the interaction between users and GitHub Copilot. Although friction is frequently referenced in Human-Computer Interaction (HCI) research [3, 6, 7, 25, 28, 43], there is little consensus on its precise definition. In this study, we define friction as delaying the user in achieving their immediate goal with the intention of encouraging reflective thinking. This friction took the form of interface interventions only allowing slow acceptance of code completions, highlighting of AI-generated code, and inline AI feedback on users' code. We designed a controlled experiment comparing two groups, one using standard GitHub Copilot with no changes and another using a customised GitHub Copilot with built-in friction elements, to investigate how these design changes affect user behaviour and perceived cognitive effort.

Our findings show that our implementation of friction into GitHub Copilot had no significant effect on any of the types of cognitive load. Likewise, there was found no significant effect on the participants' confidence and frustration levels as well as on task progression or program compilability between the two groups. Furthermore, based on qualitative interviews, we found that user perception of friction is likely shaped by contextual factors such as time pressure or educational framing. These factors appeared to shape how participants engaged with or if they even noticed friction elements. When friction was perceived, responses were often polarised: some participants found it frustrating or disruptive, while others appreciated it as a meaningful and well-considered design intervention. Furthermore, some participants prompted the AI first and used the code without fully understanding it, only attempting to make sense of it when it failed to compile. Additionally, some of the participants regretted the way they used AI, as they encountered errors and thought they could have written easier to understand code themselves.

This paper offers the following contributions to the HCI and AI-assisted programming literature: (1) We design and implement CoFriction, a custom GitHub Copilot extension that introduces three distinct friction mechanisms to encourage reflective engagement during AI-assisted coding: slow accept of GitHub Copilot code suggestions, inline feedback, and highlighting of code copied from AI. (2) We present results from a controlled study (N = 38) examining the cognitive and behavioural effects of these friction mechanisms, finding no significant impact on cognitive load, confidence, frustration, or task performance. (3) Through thematic analysis of post-task interviews, we uncover how user perceptions of friction may be shaped by contextual factors such as time pressure and educational framing.

2 RELATED WORK

In this section, we review existing research on friction in HCI, the use of friction in interactions with AI, the performance of large language models (LLMs) in programming, both for learning and professional use, and how developers collaborate with AI systems.

2.1 Friction in HCI

Some research [3, 7, 25] explores the concept of friction. Purposely designing elements that introduce thoughtful pauses or additional steps in interaction, to encourage user reflection and improve decision-making. Recent research [4–6, 9, 28, 43] goes further, and explores how friction can be applied in interaction with AI, and support more meaningful human-AI collaboration. Unlike traditional views of friction as a problem to reduce, friction is used as a constructive mechanism to enhance user engagement, trust, and cognitive processing.

According to Kahneman [17], humans act using two systems. System 1, which is automatic, and system 2, which is used when system 1 struggles, and more focused attention and active thinking is required. Cox et al. [7] discusses introducing friction in the form of microboundaries, small changes, which can have a positive effect in switching from system 1 to system 2. The authors argue that microboundaries can help people change their behaviour to reach a goal, or to keep their behaviour in line with their values. For example, by causing a switch to system 2, microboundaries can prevent impulse purchases.

Benedetti and Mauri [3] explore friction in literature, and that friction is seen as being the opposite of seamless design, which tries to make the interaction as fast and smooth as possible. This is where friction, instead, encourages people to pause and think before acting. They propose a taxonomy differentiating between diegetic and extra-diegetic friction. Diegetic is friction embedded within the user experience, such as cookie consent popups, or restrictions on social media actions, and extra-diegetic is friction introduced outside the primary platform, like speculative design artifacts, or artistic interventions that challenge norms and provoke reflection.

Mejtoft et al. [25] compared two mobile application prototypes, one incorporating friction, and one without. They found that most participants preferred the version with friction, reporting greater satisfaction, as it provided a clearer understanding of the task's goal, and encouraged deeper engagement.

Inan et al. [43] explore the use of friction in AI interactions. Making deliberate design choices that slow down the interaction, to encourage user reflection and critical thinking. Examples of such friction include prompting users with clarifying questions, or alerting them when their request matches a broad set of options. For instance, the AI may inform the user that their query corresponds to a large number of potential results, prompting them to refine or narrow their request before proceeding. They found that this friction improved task success rate, and gave the AI better insight into users' goals and beliefs.

Zeya Chen and Ruth Schmidt [6] introduce a behavioural model for friction in human-AI interaction. The model can be used for mapping different types of friction, and in their study, they identify several different real-world examples of friction, and map them to their model. The paper explores how friction can help prevent automatic or biased behaviours, increase self-control, reduce impulsive actions, and support decision-making. They identify friction as beneficial not only for AI users but also for AI developers, helping to further improve AI models. The authors highlight the importance of friction, with emerging research supporting that Human-AI interaction becomes more than user and tool, but instead a collaboration, complementing each other.

Naiseh et al. [28] conducted a within-subjects study with medical practitioners, to examine how introducing friction in interface design influences trust and cognitive engagement when interacting with AI-generated prescription recommendations. Participants interacted with a mock-up AI tool that provided both a prescription recommendation and an accompanying explanation. The study compared two nearly identical interfaces: one frictionless, and one incorporating friction through a pop-up window, requiring participants to confirm they had read the AI's explanation before proceeding. The frictionless interface was used first, followed by the friction interface eight months later. Results showed that the presence of friction increased the likelihood that participants would read the AI explanation. However, among participants who did choose to read the explanations, there was no significant difference in the amount of time spent reading between the two conditions.

Jong et al. [9] found that AI giving a partial explanation helps reduce overreliance on AI. Overall accuracy, however, was better with full AI explanations, as this reduction in overreliance also causes users to trust the AI less in cases where it was correct. Furthermore, users also preferred full explanations over partial explanations, as partial explanations required more cognitive effort, and took users longer to reach a conclusion. The authors propose that implementing partial explanations could be beneficial in systems where the AI making a mistake is detrimental, for example in medical systems.

Buçinca et al. [4] similarly explore how to reduce AI overreliance. They introduce an additional layer, requiring users to click an additional time to get AI explanations, or put in a 30-second delay where a picture pretends to load. They had similar results as Jong et al. finding that it helps reduce AI overreliance, but users preferred having the explanations as normal, without additional layers. They also found that participants' overall performance did not seem to improve. For practical use, the authors recommend deploying the interventions based on AI uncertainty and individual user traits.

Cabitza et al. [5] proposes a framework to measure how AI affects human decision-making. The framework includes a decision table that measures participants' decisions without AI, the AI's decision, and the participants' final decision after seeing the AI's decision. The authors use both a frequentist and a causal approach, to measure how the AI affects human decision-making. They conducted four medical case studies, where they found that users initially agreed with the AI's answer, but once they learned the AI agreed with them, they tended to change their answer, exhibiting algorithmic aversion. Furthermore, they also highlight the whitebox paradox, where detailed explanations from AI increase user overreliance on AI, even in cases where the AI is mistaken. This study, rather than evaluating model performance, focuses on AI models' cognitive effects on people.

These studies highlight how friction has the potential to positively influence behaviour, reducing overreliance on AI and promoting active critical thinking.

2.2 Programming in Collaboration with AI

Plenty of studies [15, 16, 20, 23, 34, 42] have explored the usage of AI in learning environments. All these studies compared how novice programmers performed with AI in introductory courses. Prather et al. [34] found that the majority of students were able to solve the task, but that there was a divide between students. Students that did not struggle were able to leverage AI effectively to help create the solution they had already planned, and were able to ignore unhelpful or incorrect code. Students who struggled, and had worse grades, were more likely to rely on the AI for solutions, and copy code without properly understanding it, creating an illusion of competence. Similarly, Zviel-Girshin [42] found that some advantages of AI were that AI could be a helpful tool to guide students if they were stuck on a problem, as well as giving them a good understanding of how to use AI for the future. One of the main downsides identified was that students might become too dependent on solutions provided by the AI. Meaning they do not get the proper understanding of coding, and are unable to code independently.

Jonsson and Tholander [15] found that students perceived the AI coding tool as both helpful and unpredictable, often requiring trial-and-error prompting to achieve desired results. While this open-ended interaction encouraged experimentation and deeper engagement, particularly through debugging, it also lead to frustration, incorrect mental models, or overreliance on the AI. Viewing AI as a precise tool versus a collaborative partner highlights a shift from mastering code syntax, to learning how to communicate effectively with the AI.

Kazemitabaar et al. [20] created CodeAid, and Liffiton et al. [23] created CodeHelp, two similar LLM-powered tools to facilitate learning by supporting students without directly providing code solutions, to avoid over-reliance on AI. Kazemitabaar et al. [20] identified four design considerations based on the feedback they got from students and educators: exploit AI's strengths, simplify input but challenge thinking, guide learning without giving away answers, and ensure transparency and user control. Liffiton et al. [23] similarly also evaluated student and educator feedback. CodeHelp was generally well received as an effective learning supplement. However, some concerns emerged, including the AI introducing concepts not yet covered in the curriculum, the risk of students becoming overly reliant on the tool, and instances of confusing responses, particularly when students were unsure how to phrase their questions.

Jury et al. [16] created WorkedGen, an LLM-powered tool for learning similar to CodeAid and CodeHelp. A key distinction, however, is that the support was delivered by breaking down solutions into step-by-step explanations. Students could click on each step to receive more detailed insights, both for the entire line of code and for specific keywords. WorkedGen was evaluated both by students and experts, with experts deeming 71% of the solutions having meaningful step separation, and 87% of students thinking the examples were separated into logical steps. Experts evaluated that 93% of the generated examples on novice to intermediate programming questions had clear explanations. 77% of students found WorkedGen helpful, and 72% of students would use it again. One identified downside was that, similar to CodeHelp, WorkedGen would sometimes include concepts students had not yet been taught, or had any pre-existing knowledge about.

These studies highlight that AI can be a useful, supportive tool for learning if used in a well-thought-out way, but it also poses a risk that students might become too reliant on AI, and unable to write code and think of creative solutions on their own.

Studies [13, 14, 26, 29, 30] have explored the capabilities of LLMs such as GitHub Copilot and ChatGPT in software development tasks, often comparing them to human programmers. These studies highlight the potential of AI tools, but also their limitations, suggesting that AI functions best as a collaborator rather than a replacement for human developers. Nguyen and Nadi [30] evaluated GitHub Copilot and the quality of its code suggestions through an empirical study. They found that the correctness rate of its suggestions varied based on the code language. The study also found that GitHub Copilot's suggestions have low complexity, indicating that the suggestions are understandable. It also found some shortcomings with suggested code that could be simplified further, or suggested code that relies on undefined helper methods. Similarly, Dakhel et al. [26] also studied the quality of GitHub Copilot's suggestions, and how it compared to human programmers. They found that Copilot is able to generate correct and optimal solutions for fundamental problems, but that the quality of the code also heavily depends on the quality of the prompt. They also found that GitHub Copilot could suggest more advanced solutions than junior developers, while having similar correctness to a human solution and similar technical issues. However, an experienced developer is required to evaluate the solution and filter out bugs and non-optimal solutions, making GitHub Copilot a great collaborative tool for an experienced programmer, but riskier for newer programmers. Nascimento et al. [29] did an empirical study comparing ChatGPT and programmers when solving LeetCode tasks in C++. Their results seem to support the findings of Dakhel et al. They found that ChatGPT's solutions consistently surpassed novice coders' solutions, but expert coders' solutions surpassed ChatGPT, and were also able to solve tasks in which ChatGPT struggled.

Idrisov and Schlippe [14] benchmarked several AI models with 6 LeetCode problems, each AI generating 18 program solutions. They found that only 26 out of the total 122 generated solutions were correct, though an additional 11 of these incorrect solutions would be valid with minor edits. They also found that for these 11 incorrect solutions, it was between 8.85% and 71.31% faster to correct these solutions rather than create a solution from scratch. Among the tested models, GitHub Copilot, powered by Codex (GPT-3.0), achieved the highest task success rate, creating 9 correct programs, while ChatGPT (GPT-3.5) only made 4. Interestingly, however, ChatGPT was the only model that successfully generated a correct solution for a hard problem. BingAI Chat, despite being powered by GPT-4.0, was not able to solve a hard problem, though it created 7 correct solutions. These results highlight the variation in performance across models, and further support the view that current AI tools, while promising, still often require human validation or refinement, especially on harder problems.

Huang et al. [13] created an LLM debugging prototype called ChatDBG. They evaluated it in Python and C/C++ using different configurations. They found for Python a single query would lead to a successful bugfix 67% of the time, and a follow-up query would increase this to 85%. For C/C++ programs, the root cause of a problem could be identified 36% of the time, and the proximate cause could be found an additional 55% of the time. They overall found that engaging in dialogue with ChatDBG significantly assists developers in debugging.

These studies demonstrate that, while AI can generate correct and sometimes optimal code, it lacks the nuanced understanding and judgment of experienced developers. This performance gap supports the view that human oversight and collaboration are essential when using AI for programming tasks.

Expanding on this, other studies [8, 12, 31, 32, 35] investigate how developers are beginning to collaborate with AI. Russo [35] looks at how software engineers are starting to use Generative AI tools, like LLMs, and what factors shape their adoption. The study combines surveys and interviews, and introduces the Human-AI Collaboration and Adaptation Framework (HACAF) to explain how individual, technological, and social factors interact in this process. One key finding is that compatibility with existing workflows matters more than traditional factors like perceived usefulness or peer influence. Russo also highlights common concerns, such as job security, data privacy, and code quality, that can slow adoption and offers practical suggestions for how teams can better support AI integration. This could include managers ensuring employees that AI is just meant as a supportive tool, not a replacement, and addressing code quality concerns.

Hamza et al. [12] did a workshop evaluating how software engineers collaborated with ChatGPT for writing code. The evaluation was done through responses to open-ended questions and observations during the workshop. The workshop showed that the participants found ChatGPT useful for being a collaborative partner that could serve different creative roles and help design the solution, rather than exclusively being a tool for writing code. They also identified some potential concerns, including a risk that the proposed AI solutions are not made with up-to-date coding practices in mind and might include security risks. The AI also struggles when having to create particularly complex solutions, and the programmer might become too reliant on the AI to create solutions without properly ensuring the quality of the solution.

Peng et al. [32] did a controlled study and found that GitHub Copilot allowed programmers to finish tasks 55.8% faster than the control group. They also found indications that less experienced programmers benefited more from the help of GitHub Copilot. Cui et al. [8] evaluated generative AI's effect on productivity through random controlled trials at Microsoft, Accenture, and a third anonymous Fortune 100 company. They found that AI increased developer productivity by 26.08%, and, similarly to Peng et al. observed that less experienced developers had a more significant productivity increase and higher adoption rate.

Panchanadikar and Freeman [31] investigated how AI affects the field of indie game development through a qualitative analysis of 3091 online posts and comments from Facebook posts and different subreddits. They found some advantages of AI were that indie developers can create assets that would normally be too expensive, and it simplifies the process for developers without technical expertise to get more into the field. It can also help the creative process and serve as a useful social companion, providing support and mentorship in a field that can get very lonely. They also found certain risks and concerns. Lack of originality was a concern due to the way generative AI currently works, which might cause all assets between games to look similar and make an original design difficult. There were also just general career concerns with AI reducing the need for human developers in game development. Having to train a generative AI model that might fit the developer's specific needs would be time-consuming and might pose a risk for the limited time and budget indie developers have. Due to these concerns, the authors propose three design principles for AI: let AI be more customisable, highlight the importance of human-AI collaboration, but also make AI facilitate human-human collaboration, and have AI serve as an entity for generating content and an entity for providing emotional support.

These findings further support that LLMs can enhance developer performance as a collaborative tool, rather than serving as a replacement for human programmers.

2.3 Intentional Friction in Developer-AI Interaction

Kazemitabaar et al. [19] conducted two studies where they introduced deliberate friction in the interaction with AI, which slows down interaction to create a deeper cognitive engagement for users while coding. They evaluated seven different cognitive engagement techniques. They found that the technique 'Lead and Reveal', which required users to answer questions about each step before revealing the corresponding code, improved users' self-assessment accuracy without increasing cognitive load, which has the potential to improve learning and prevent overreliance on AI. Their work is closely related to ours, particularly in its exploration of friction-based interventions and cognitive load. Notably, several of their techniques, including 'Lead and Reveal', overlapped with ideas we had independently considered.

However, we chose not to pursue such approaches due to their higher levels of intrusiveness, which we believe makes them less suitable for general programming scenarios outside of structured learning environments. While Kazemitabaar et al. treat stable or reduced cognitive load as a positive outcome, we instead hope to see an increase in cognitive load due to friction, potentially indicating greater user reflection or critical engagement with AI-generated code. Rather than aiming to draw strong conclusions about outcomes, our study remains exploratory, investigating whether such friction has any measurable effect on cognitive load in more traditional programming settings.

3 METHOD

To investigate how intentional friction affects user interaction with AI programming tools, we conducted a controlled, between-subjects experiment comparing a standard version of GitHub Copilot to a customised version, CoFriction, that introduces friction mechanisms into the interaction with the AI. Our goal was to assess whether such interventions influence cognitive load, confidence,

frustration, and task performance during programming. We collected both quantitative and qualitative data through programming tasks, an experience survey, an expanded cognitive load questionnaire, and post-task interviews. This section outlines the experimental design, participants, materials, procedure, and analysis methods used in the study.

3.1 Cognitive Load Theory

Cognitive Load Theory (CLT) is a psychological model that describes how the human brain processes and stores information, particularly in learning environments. Developed by John Sweller [39] in the late 1980s, the theory initially described what would later be termed intrinsic cognitive load (ICL) and extraneous cognitive load (ECL). In 1998, Sweller et al. [40] expanded the model by introducing the concept of germane cognitive load (GCL), thereby distinguishing three types: ICL, ECL, and GCL.

- **ICL:** the inherent difficulty of the material. It depends on the individual's prior knowledge.
- **ECL:** the way information is presented, which can either help or hinder learning. Poor instructional design increases extraneous load. Generally, this should be reduced to maximise learning.
- **GCL:** the effort devoted to processing, constructing, and automating mental schemas. This is where actual learning is measured.

Klepsch et al. [21] introduced a differentiated approach to measuring ICL, ECL, and GCL across two studies. In the first study, participants were divided into two groups: one received a basic introduction to CLT, while the other did not. Both groups were asked to evaluate 24 different learning scenarios. The informed group assessed the scenarios based on their CLT knowledge, whereas the uninformed group completed a questionnaire consisting of seven items rated on a 7-point Likert scale, two items each targeting ICL and GCL, and three targeting ECL. The informed group accurately distinguished between high and low load scenarios and identified the type of cognitive load involved. The uninformed group performed well in distinguishing ICL and ECL, but struggled with GCL. This suggested that while an informed group could offer valid results, the requirement for prior training limited practical applicability.

To address this, a second study was conducted using only uninformed participants. The questionnaire was revised, including an additional item for GCL, recommended for use only when GCL is varied on purpose. Results indicated that the modified instrument enabled participants to reliably differentiate between high and low load across all three load types. These findings from the second study inform the development and interpretation of our own questionnaire used to assess participants' cognitive load in our experiment. While CLT is mainly used in educational settings for learning, it also offers a useful framework for measuring cognitive load in a broader sense, which suits the aims of our experiment.

3.2 CoFriction: Custom Copilot Extension

Our extension is a customisation of GitHub Copilot, it implements three friction elements to cover the use cases of Copilot to ensure the users encounter friction. The three friction elements are: slow accept of Copilot's code suggestions, inline feedback from AI, and highlighting of code copy-pasted from the AI chat. A screenshot of CoFriction in action can be seen in Figure 1.

Similar to prior work [33], we consider slow accept to refer to the process where users manually type out Copilot's suggestions instead of accepting them with the press of a key.

We disable the ability to accept suggestions, thereby if users want to accept a Copilot code suggestion they have to slow accept by typing them out. An example of a GitHub Copilot code suggestion can be seen in Figure 2a.

In Figure 2b, an example can be seen of a user manually writing a code suggestion while the code is displayed.

Slow accept of Copilot's code suggestions is implemented locally for our experiment. It consists of editing the default keybindings.json file in Visual Studio Code to disable the keybinds used to accept suggestions. Furthermore, a .vscode folder with a settings.json file is added inside the folder the participant uses for their code. The settings.json file disables the toolbar that shows when hovering a Copilot suggestion. This toolbar normally serves as an additional way to accept Copilot suggestions.

Inline feedback is a new feature we designed by prompting GPT-4o the user's code and asking for line-by-line feedback to the lines where the AI deemed it severe enough to impact readability and maintainability of the code. The AI feedback is then added at the end of the line it belongs to. To avoid taking up the entire screen only 25 characters of the feedback is visible by default. An example of this can be seen in Figure 3a. The feedback can be seen in full by hovering the feedback, as can be seen in Figure 3b.

To avoid sending too many request at once, a request is only sent if 3 seconds pass without any document edits. Depending on the size of the document, there is a delay before all new feedback shows up, as there is some processing time and the feedback has to be applied to the relevant lines. If there is inline feedback on the line the user is currently editing, the feedback is removed.

Code that has been copied from the AI chat is highlighted, as long as the copied code is more than 4 characters. This means the whole code snippet does not have to be copied to highlight the copied code. The colour chosen for the highlighting matches the highlighting of selected text, an example of the highlighting can be seen in Figure 4.

When checking for code to highlight all indentation is ignored in both the user's code and the AI generated code. The AI chat history is only available when the user tags our chat participant by starting their message with '@tutor'. To only highlight copied code we look for code start tags and end tags in the AI chat. Text in between a pair of start tag and end tag is code and checked against when something is pasted into the file.

Code that has been copied from the AI chat is added to an array of copied code snippets. This way the code will be highlighted even after the code has been moved around. Making changes to the highlighted code will remove the highlighting.

Inline feedback and code highlighting for copy-pasted AI code were implemented as part of a Visual Studio Code extension and uploaded to the extension marketplace. Meaning two of our friction types are accessible by simply downloading this extension, but our slow accept friction requires local modifications.

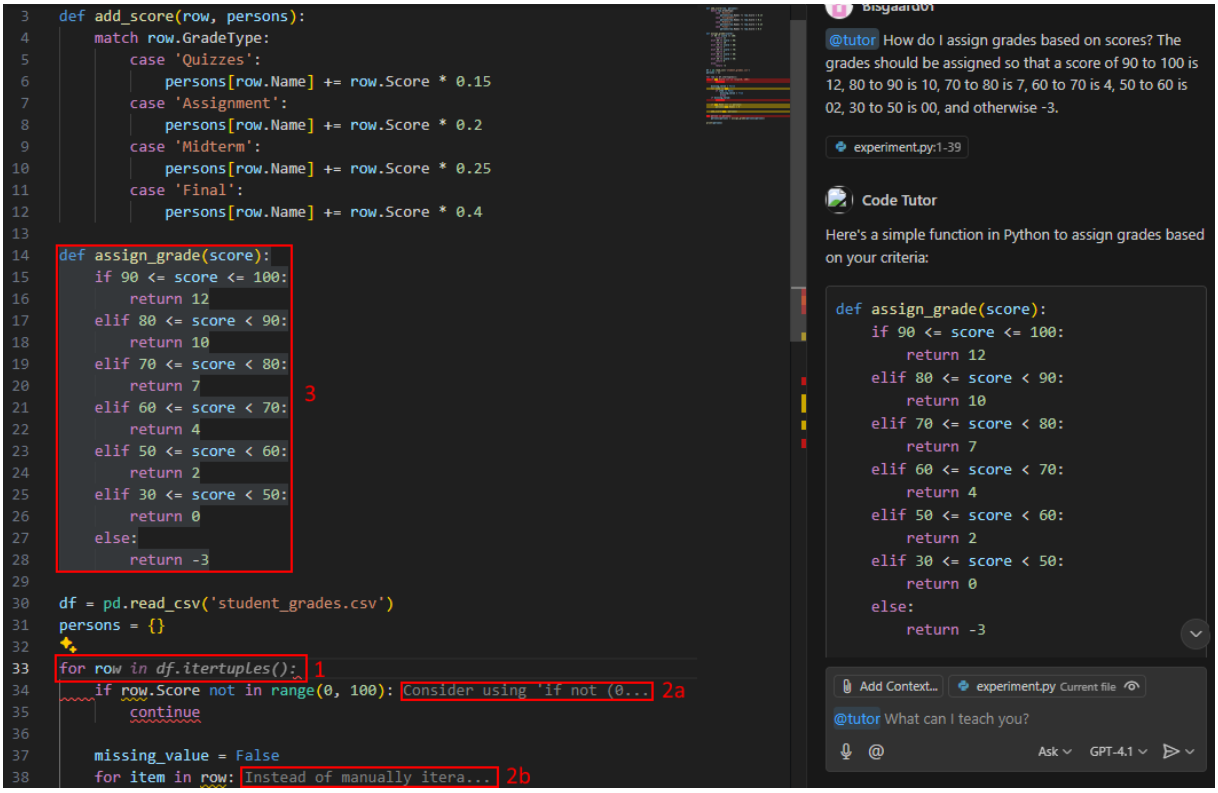
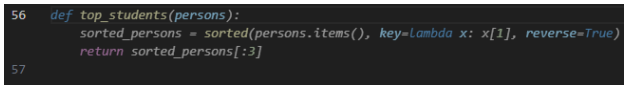


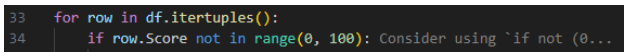
Figure 1: Screenshot of CoFriction in action. 1) slow accept of a Copilot code suggestion. 2a) and 2b) inline feedback from AI. 3) highlighting of code copy-pasted from the AI chat on the right.



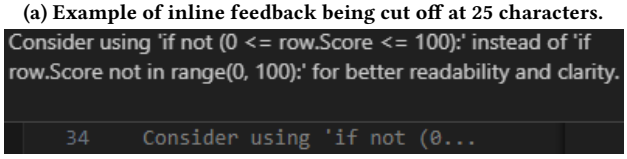
(a) An example of a GitHub Copilot code suggestion.

(b) An example of slow accepting a GitHub Copilot suggestion by typing it out.

Figure 2: Examples of slow accept for Copilot suggestions.

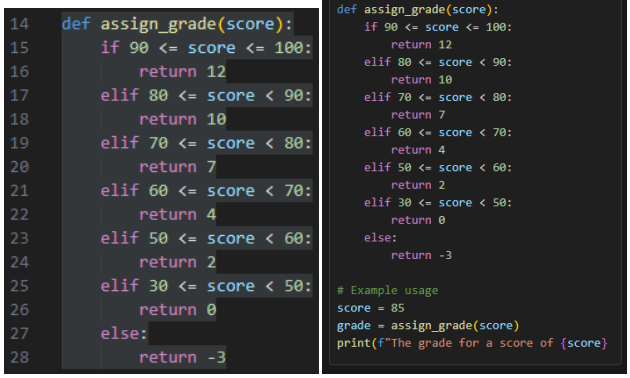


(a) Example of inline feedback being cut off at 25 characters.



(b) Example of hovering the inline feedback to see the full feedback.

Figure 3: Examples of inline AI feedback.



(a) Highlighting of copied code. (b) AI chat window containing the copied code.

Figure 4: Example of highlighting of copy-pasted code from the AI chat.

3.3 Materials

To evaluate the effect of introducing friction in AI-assisted programming, we used a combination of scripted facilitation, surveys, programming tasks, post-experiment interviews, and a custom cognitive load theory, frustration, and confidence (CLTFC) questionnaire. This allowed us to collect both quantitative and qualitative data on participant experience and perceptions. The following subsections detail each material used in the study.

3.3.1 Facilitator Script. To ensure consistency between all participants, we used a standardised script throughout the experiment, which the facilitator would follow. The facilitator would follow the script as closely as possible, however, deviations can occur if participants have questions or did not follow the instruction as intended. The full script can be found in Appendix A and its usage is reflected in Section 3.6, which explains the procedure of the experiment.

All participants were guided using the same script, but the friction group were additionally told to include an '@tutor' tag in their message when prompting the AI, as this was necessary for the functionality of our extension, as explained in Section 3.2.

3.3.2 Experience Survey. Participants had to fill out a survey before the experiment could begin. The survey gathered demographic and experience-related data: age, general programming experience, Python experience, and prior use of AI in programming. The survey can be found in Appendix B.

The purpose of collecting this data was to provide a demographic context for the results. For instance, individuals with advanced proficiency in Python and familiarity with Copilot may experience the tasks and work environment with reduced frustration, irrespective of their assignment to the friction or non-friction group.

3.3.3 Programming Tasks. The overall objective of the tasks were to create a Python program for importing and analysing student grades, as well as generating student reports. The full task description can be found in Appendix C.

The programming tasks were designed to be both numerous and simple, with the goal of having the participants interact as much as possible with the Copilot tools, i.e., we wanted participants to produce a lot of code with the help of Copilot and CoFriction. If there were too few tasks, participants may have solved them too quickly, lowering time spent with the extension, and if the tasks were too complex, they may get stuck without even Copilot being able to assist them. Furthermore, the tasks followed a progressive structure, where participants had to use previous tasks to be able to solve the next one.

3.3.4 CLTFC Questionnaire. To measure cognitive load, we adapted the questionnaire items from the Klepsch et al. CLT study [21] (see Section 3.1), making minor modifications to suit the specific context of our experiment. As noted in the original study, such adaptations are appropriate when applying the instrument to new domains.

To gain additional insight into participants' experiences, we included two supplementary questions targeting their confidence and frustration levels. We chose this instrument for its proven reliability in measuring cognitive load.

Our additional questions on confidence and frustration were included to capture more affective dimensions of the participant experience, which are not directly covered by the original CLT questionnaire. The complete questionnaire used in the experiment is provided in Appendix D.

3.3.5 Interview Questions. At the end of the experiment, the participants were interviewed. Interviewing the participants provides qualitative data, offering deeper insights into their perspectives on using AI for programming, the concept of introducing friction, and our specific implementations of friction. The full interview can be found in Appendix E.

Both non-friction and friction participants were asked a baseline set of questions. These questions aims to explore participants' views on the general use of AI for programming and to assess whether they felt the AI had assisted them during the experiment.

Following this, participants were presented with questions specifically designed for their group: friction or non-friction.

3.4 Setup

To ensure consistency and minimise external influences on participant performance, the experimental setup was carefully designed and standardised across all sessions. This section outlines the physical and digital environment in which the study was conducted, including details of the room configuration, equipment, and software setup.

3.4.1 Environment Setup. The experiment was conducted in a quiet, private room at Aalborg University, designed to minimise distractions and support focus during programming tasks. The setup is illustrated in Figure 5. The setup consisted of a standard Windows 11 laptop equipped with a wireless mouse and mousepad, placed on a corner-desk. The participant was seated at the desk, with the laptop in front of them for programming tasks and moved aside for paper-based activities, such as completing the consent form and our CLTFC questionnaire (see Section 3.3.4).

The facilitator sat approximately 1 meter to the left of the participant to monitor progress and provide assistance if requested, e.g., for IDE or Copilot/CoFriction issues. Following a standardised script, the facilitator delivered task instructions, managed timing, and distributed paper materials, addressing participant questions only about the setup or tools. Only the participant and facilitator were present during the experiment to reduce potential distractions.

3.4.2 Laptop Setup. The software environment was configured on a laptop running Windows 11 with a hybrid ANSI-ISO Nordic keyboard layout. Visual Studio Code (version 1.99.3) was used as the IDE for the experiment, with Python 3.12.

The laptop had certain packages commonly used for data processing pre-installed, e.g., pandas, and participants were free to install new packages if needed.

3.5 Participants

All participants were recruited through our extended network, and were compensated with a piece of candy.

For the recruitment, we had the following inclusion criteria:

(1) Software Engineering Master's student
(2) Prior experience with AI-assisted programming
(3) Prior experience with the Python programming language

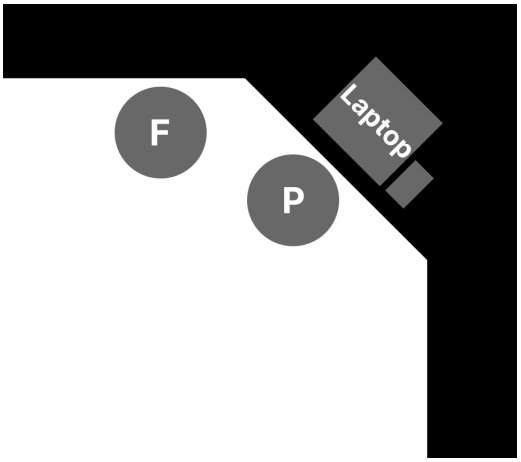


Figure 5: High-level figure of experiment setup. P is participant and F is facilitator.

The basis for inclusion criterion 1 was primarily pragmatic: Master's students in Software Engineering were readily accessible through our academic network, and focusing on students from the same degree program helped streamline prior participant experiences. Although we initially considered full-time professional programmers, who often work on production-level code that may be more directly impacted by AI-assisted tools, we determined that recruiting a sufficient number of professionals was not feasible for this study. To ensure the students had relevant background experience, we included only those with any prior experience in both programming with AI tools (criterion 2) and in using the Python programming language (criterion 3). This was deemed essential, as the experiment investigated AI-assisted programming workflows in Python. By ensuring a basic familiarity with both, we aimed to reduce the influence of confounding factors and better isolate the effect of friction.

3.6 Procedure

Participants were greeted upon arrival and asked to take a seat. They received a written consent form detailing their rights and data usage, which they signed before proceeding.

Participants completed the programming background survey described in Section 3.3.2. The facilitator then introduced the Python programming tasks (see Section 3.3.3) and explained the procedure. The workspace consisted of Visual Studio Code, with the control group using the standard Copilot extension and the friction group using a custom Copilot extension designed to introduce usability challenges (see Section 3.2). The facilitator instructed participants that they had to execute their code using the integrated terminal within the editor. Participants were informed they could request assistance with the IDE or Copilot and that completing all tasks within the 25-minute time limit was not required.

When ready, participants started a timer and began the tasks. The experiment ended when participants completed all tasks, the time expired, or they chose to stop.

Evaluation began with our CLTFC questionnaire, measuring cognitive load, frustration, and confidence, see Section 3.3.4. A semi-structured interview followed, with responses recorded for transcription. The interview included general questions about AI and using AI when programming, as well as, additional questions based on the participant's group, friction or non-friction, see Section 3.3.5.

The experiment concluded with participants offered the option to learn that the study focused on cognitive load under friction conditions, which was initially withheld to avoid priming. Lastly, there were offered a piece of candy and were bid farewell.

4 RESULTS

We recruited 38 male Software Engineering Master's students, with a mean age of 24.6 years (SD = 1.4). They were divided into two groups: friction and non-friction, with 19 participants in each. An overview of the demographics for the two groups can be seen in Table 1. Before the experiment, participants were required to answer a survey assessing their programming background. The survey can be found described in Section 3.3.2. Participants had a mean general and Python programming experience of 6.3 and 2.4 years, respectively. Most participants (66%) have programmed in the contexts of work, of which nearly half (48%) have used Python. The majority of participants (89.5%) reported at least occasional use of AI coding tools, with 52.6% indicating frequent usage ('Often' or 'Always'). When asked about how they use AI tools for coding, participants most commonly reported using them for debugging (79%) and learning new concepts or syntax (68%). Other common uses included code completion (53%) and generating entire code snippets (53%). A few participants (N = 4) listed other specific use cases.

	Friction	Non-friction
Number of participants	19	19
Age range	23-26	22-30
Mean age (SD)	24.4 (0.8)	24.8 (1.9)
Mean general experience (SD)	6.2 (1.5)	6.5 (2.0)
Mean python experience (SD)	2.8 (1.6)	1.9 (1.7)

Table 1: Demographics for the two study samples. All values expect number of participants are in years.

4.1 Cognitive Load

A Friedman rank sum test indicated a significant difference between the different types of cognitive load ($\chi^2(2, N = 38) = 16.36, p < .01$). Post-hoc pairwise comparisons using Conover's test with Bonferroni correction indicate that GCL is significantly higher than both ICL and ECL, both $p < .01$. The total distribution of the cognitive load scores, sorted by group, can be seen in Figure 6.

Next we look at the cognitive load scores between the friction and non-friction group. There was no significant effect for ICL $t(36) = 0.20, p = .84$, GCL $t(36) = 0.79, p = .44$, and ECL $t(36) = 0.57, p = .57$ despite the friction group having lower cognitive loads: ICL ($M = 3.32, SD = 1.16$), GCL ($M = 4.53, SD = 1.27$), and ECL ($M = 2.84, SD = 1.07$) than the non-friction group: ICL

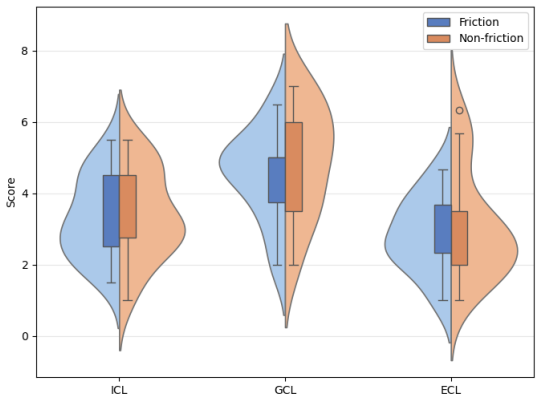


Figure 6: Cognitive load distributions for every participant split by cognitive load type and friction group.

($M = 3.37, SD = 1.26$), GCL ($M = 4.89, SD = 1.59$), and ECL ($M = 3.09, SD = 1.52$).

To investigate whether friction and participant characteristics influenced outcomes, a series of two-way ANOVAs were conducted. These analyses assessed the effects of friction in combination with AI usage, general programming experience, and Python experience on five dependent variables: ICL, GCL, ECL, confidence, and frustration. Each outcome variable was tested in three separate models, 15 total, with no significant main effects or interactions emerging, except one.

For ECL, a significant main effect of general experience was found ($F(8, 22) = 3.01, p = .019, \eta^2 = 0.52$). However, there was no significant main effect of friction ($F(1, 22) = 0.47, p = .50, \eta^2 = 0.02$) and no significant interaction between friction and general experience ($F(6, 22) = 0.88, p = .52, \eta^2 = 0.19$). This suggests that participants' general experience was associated with differences in ECL regardless of friction condition.

To further explore the significant main effect of general experience on ECL, a polynomial regression analysis was conducted. A model including both linear and quadratic terms provided a significantly better fit than the linear model alone ($F(1, 35) = 8.54, p = .006$). This indicates a significant quadratic relationship between general experience and ECL.

Specifically, ECL was lowest at 5-8 years of general experience, and higher at 4-4.5 and 10 years of general experience, forming a U-shaped curve, with both less and more experienced participants reporting higher levels of ECL.

4.2 Frustration, Confidence & Tasks

Furthermore, we look at frustration and confidence levels for the friction and non-friction groups. No significant effect was found for both confidence $t(36) = 1.06, p = .29$ and frustration $t(36) = 0.17, p = .87$. The friction group had a similar frustration score ($M = 3.53, SD = 1.78$) to the non-friction group ($M = 3.53, SD = 2.01$), whereas the confidence score was lower for the friction group ($M = 3.05, SD = 1.87$) than the non-friction group ($M = 3.53, SD = 2.27$).

Additionally, we look tasks started for the friction and non-friction groups. There was found no significant effect for tasks started $t(36) = 1.39, p = .17$. The friction group started fewer tasks ($M = 4.32, SD = 1.34$) than the non-friction group ($M = 5.32, SD = 2.85$).

Lastly, we look at the tasks started that compile and the tasks that do not compile for the friction and non-friction groups. In the friction group 12 solutions compile and 7 do not compile, and for the non-friction group 13 of the solutions compile and 6 do not. There was found no significant effect for the amount of tasks started when the code can compile $t(23) = 0.92, p = .37$. Similarly, no significant effect was found for the amount of tasks started when the code does not compile $t(11) = 1.07, p = .30$. The friction group started more tasks when the code could compile ($M = 4.5, SD = 1.17$) compared to when the code could not compile ($M = 4.0, SD = 1.63$). Whereas, the non-friction group started fewer tasks when the code could compile ($M = 5.1, SD = 2.2$) compared to when the code could not compile ($M = 5.8, SD = 4.2$).

4.3 Participant Reflections

As described in Section 3.6, participants were asked a series of semi-structured follow-up questions after completing the programming tasks. These questions aimed to elicit participants' subjective experience, perceived challenges, and evaluative reflections on using CoFriction and Copilot.

To synthesise the diverse qualitative responses, we conducted a thematic analysis. The idea was to group the reflections into themes that highlight recurring patterns, points of tension, and notable individual perspectives across participants.

The thematic analysis was conducted collaboratively, as all authors were closely involved with the material, either by facilitating the interviews or by transcribing them, which gave each of us a deep familiarity with the content. We engaged in joint discussions to identify recurring topics and patterns that stood out across the interviews. Through this collaborative interpretation, we agreed on a set of common themes that captured the core ideas expressed by participants. We then selected and translated representative quotes from the transcripts to illustrate and support each theme. Although our approach was not strictly formalised, it followed the key principles of thematic analysis by centring on shared meaning and pattern recognition across the qualitative data. This approach allowed us to distil common concerns and insights, while still preserving the variability in how different users experienced the tools.

The following subsections outline the six themes that emerged from this analysis. These themes reflect a mix of cognitive, emotional, and practical responses to both the AI assistant and the friction elements embedded in the CoFriction interface.

4.3.1 Theme 1: User Experience With Friction. Many participants expressed positive thoughts on friction for the purpose of enabling programmers to think more about what they write. The most often praised was slow accept. *'You definitely end up thinking more about what you are coding when you have to write it all out, rather than just auto-completing it.'* ([P19], friction group). However, an equal amount of participants expressed frustration with the slow accept

friction element. *'It was mostly just frustrating. I like being able to tab [auto-complete].'* ([P5], friction group).

However, some participants also found the base Copilot auto-complete functionality to be distracting: *'I think it [auto-complete suggestions] can end up being a bit disruptive, because you're not the one planning things yourself.'* ([P23], friction group). Others found it stressful: *'… except for that tab thing [auto-complete], that stressed me out.'* ([P36], non-friction group). Or even frustrating: *'I found it frustrating that it [auto-complete suggestion] kept popping up with all sorts of things – like, I have some idea, and I'm in the middle of writing it, and then it comes up with something else, and then I'm taken away from the idea I originally had. I find that annoying.'* ([P33], friction group).

Few participants also expressed mixed feelings about the friction, admitting it was frustrating, but at the same time believed that it could potentially be beneficial anyhow. *'It was frustrating … but it might be smart in the long run if you're new to it.'* ([P7], friction group).

Despite frustrations, some participants suggested that the nature of the task could be decisive in whether or not they find friction useful: *'It felt okay, but if you had to solve something real, it would be distracting.'* ([P27], friction group).

Participants were the most expressive about the slow accept friction, while the other two friction elements, inline feedback and copied code highlighting, often went unnoticed by them: *'I actually had not noticed that.'* ([P19], friction group) when being asked about code highlighting. And in other cases, the friction was deliberately ignored: *'I didn't read it. I mean, I did see it was there, I just thought it was a bit annoying.'* ([P15], friction group) when discussing inline feedback. With some perceiving the elements as obstructive rather than beneficial: *'It is more disturbing than it helps.'* and *'I would actually prefer it did not do that.'* ([P27], friction group) when discussing code highlighting and inline feedback, respectively. However, a number of participants also responded positively to the idea of code highlighting and inline feedback in the interview, even if they did not really interact with or notice it during the task. Several participants mentioned that highlighting could be useful for knowing what code they should read more thoroughly to make sure it is working as intended: *'I think it makes a lot of sense, primarily because now I imagine a prototype environment where 'tainted' code is marked so then you remember you should probably give it a look again at some point.'*([P31], friction group). *'I like it cause then it is directly marked which code is AI generated and which code is written by me.'* ([P17], friction group). A few participants responded very positively to inline feedback: *'I did not notice that. That is actually really nice. I actually did not notice that at all. I like it, I actually really like it.'* ([P29], friction group). *'I thought that was really cool. Normally, you just get those next word predictions. It was actually interesting to get something like a 'hey, try to consider this.' That was actually really cool.'* ([P7], friction group). However, As mentioned earlier, several participants did not like code highlighting and inline feedback or had no opinion on it, as they did not notice it during interaction.

4.3.2 Theme 2: Time Pressure. Participants generally felt like they were under time pressure while solving the tasks, despite explicit instructions clarifying that they are not expected to be able to solve all the tasks within the allotted time. This perception was shared across both groups: *'I kind of felt like I had to hurry even though I knew it did not matter whether I finished or not.'* ([P1], friction group) and *'Yes, definitely. Because of the time limit.'* ([P20], non-friction group).

This perceived time pressure appeared to influence how participants interacted with our friction implementations. Specifically, several participants indicated that they prioritised speed over engagement with the inline feedback: *'If I didn't have a time constraint, then I would probably look more at the inline feedback. But to rush through the assignments as quickly as I wanted, I didn't really look at it that much.'* ([P9], friction-group).

4.3.3 Theme 3: Relearning Python With AI. Many participants explained increased AI-chat use due to their need to reacquaint themselves with Python syntax, which they had forgotten due to recent inactivity: *'I haven't coded in Python for a while, so I used it more than I usually do – mostly to get help with the syntax.'* ([P4], non-friction group). Another participant similarly noted the utility of AI in refreshing Python knowledge: *'I haven't programmed in Python for a while, so I was more inclined to reach out to it [AI] for help.'* ([P9], friction group).

4.3.4 Theme 4: Prompt First, Think Later. A few participants stated that their strategy for using the AI for programming would be to first copy-paste the AI-generated code, to see if it compiles, and only if it does not, they would try to understand the code. *'I probably spent more time trying to see if it worked, and if it didn't, then I looked into what it was doing – so yes, I did fix the errors that were there sometimes.'* ([P19], friction group). This is further reinforced by another participant: *'I just take it [AI-produced code] and try it, and figure it out afterward.'* ([P5], friction group).

This strategy illustrates the issue with relying too much on AI-generated code, where executable code is superficially implemented without deeper scrutiny, provided it does not cause immediate errors. Some had a similar approach, but did still try and remain critical depending on the complexity of the code: *'Yes, I skimmed through it quickly. If it was more complicated, I spent more time looking at it than if it was something simple.'* ([P3], friction group). Another participant chose to trust the AI's output due to its simplicity, but also because they have previously encountered similar code: *'Initially, I rushed through because they were simple lines that resembled something I'd seen before, so I trusted it greatly.'* ([P8], non-friction group).

Others acknowledged that their reactive strategy sometimes backfired, especially when it led to unresolved issues: *'I should clearly have spent more time on it … I ended up with errors that I tried to solve without fully succeeding.'* ([P4], non-friction group). Similarly: *'I usually just throw it in and see if it looks right; if not, then I try to analyse why it's wrong.'* ([P26], non-friction group).

4.3.5 Theme 5: Friction in an Educational Setting. Several participants expressed that while the friction mechanisms might be frustrating in regular workflows, they saw clear value in them within an educational context. The added effort required by slow accept and the presence of inline feedback were perceived as tools that could help learners better understand the code they work with. *'I thought it was annoying that it couldn't do it. From a learning perspective,*

it's probably a good thing that you get it into your fingers by writing everything yourself. And then you might also think more about what it's actually suggesting.' ([P19], friction group).

This sentiment was echoed by others who, despite finding the friction somewhat annoying, acknowledged its potential benefits for novices. *'If you're just starting out, it might actually help you learn more, even though it's a bit frustrating.'* ([P7], friction group).

Only few participants considered whether friction elements could be useful outside of education. For most, the value of friction seemed to hinge on its role in guiding deeper understanding rather than accelerating workflow. *'It's probably more useful for learning than for getting something done quickly.'* ([P15], friction group).

4.3.6 Theme 6: Participants Wanted to Use Less AI. Several participants expressed a desire to reduce their reliance on AI for various reasons. Some encountered difficulties getting AI-generated code to function properly and consequently preferred to solve tasks independently: *'I got a lot of errors, which I then tried to fix, but I didn't fully succeed. So yes, in retrospect, I should probably have done it from scratch.'* ([P4], non-friction group). The same participant reflected the sentiment that it was due to the feeling of time pressure: *'… I probably did, because of the time pressure.'* ([P4], non-friction group). Others also felt as if the AI suggestions were not helpful and would try to suggest implementations that did not make contextual sense: *'I felt like it didn't have enough context — or maybe I wasn't good enough at giving it enough context — so what it ends up generating is really just a bunch of things that I don't feel fit together. If I had written things myself instead, I feel like there would have been more structure, and it would have been easier to understand the code.'* ([P27], friction group).

5 DISCUSSION

In this study, we looked at implementing friction into the interaction with AI programming tools. No significant effect was found between the friction and non-friction group for any of the metrics.

In this section, we discuss factors that could have impacted the results, limitations, and propose future work.

5.1 Interpreting the Effects of Friction

Although no significant effects were observed for friction across any of the metrics, a significant main effect of general programming experience on ECL was found. The observed effect of general experience on ECL occurred regardless of friction condition, as no interaction was found. Follow-up polynomial regression analysis indicated a significant quadratic, U-shaped, relationship between experience and ECL. Participants with mid-level experience (approximately 5–8 years) measured the lowest levels of ECL, while those with either less (4–4.5 years) or more (10 years) experience measured higher levels. The reason for this pattern is unclear and could warrant further investigation.

Furthermore, as noted by Cox et al. [7], friction is often perceived as a contributor to user frustration. In our study, however, participants in the friction condition did not report significantly different levels of frustration compared to the control group. This contradicts the expectation that added friction, particularly when it interferes with fluid interaction, would increase frustration. Furthermore, since friction is intended to introduce interference, and given that ECL, as discussed by Sweller et al. [39, 40], is influenced by how information is presented, it is reasonable to expect that friction could increase ECL.

One possible explanation is that the implemented friction was too subtle to meaningfully impact participants' experience. As we defined in Section 1, friction in our study refers to deliberate interruptions that delay users from achieving their immediate goals, with the intention of encouraging reflective thinking. However, two of our mechanisms, inline feedback and code highlighting, were often overlooked or ignored by participants, suggesting that they may not have introduced enough disruption to engage users in deeper reflection. This raises the possibility that these friction types, as implemented, were insufficient to activate the reflective processes we intended. While the slow accept mechanism was noticed more frequently, it alone may not have been enough to shift user cognition in a time-constrained setting. Still, this interpretation should be treated cautiously. Other factors, such as individual differences, participant count, task familiarity, the perceived time pressure, or the framing of the study, may also have played significant roles. The absence of strong effects is likely shaped by a confluence of such variables, and further investigation is necessary to better isolate their relative influence.

Slow accept was more frequently noticed and commented on by participants. Several mentioned that the delay in accepting Copilot suggestions was frustrating. Yet, this subjective feedback did not translate into a measurable increase in frustration scores. This suggests a potential disconnect between user perception of friction and the way it affects their reported frustration, at least within the parameters of our study.

5.2 Time & Friction

Our experiment was motivated by the concern that frictionless AI coding tools may encourage passive acceptance of suggestions, potentially leading to increased coding speed at the cost of reduced code quality and programmers' code understanding [16, 20, 23, 34, 42]. To investigate this, we introduced friction into the interaction with AI to aim to increase user engagement. Several participants in the interview mentioned they felt that the AI made them code faster, supporting the findings of Peng et al. [32] and Cui et al. [8], but a group of participants also mentioned they felt the code quality decreased because of how much they relied on the AI as mentioned in *Theme 6*.

The time pressure discussed in *Theme 2* might also have made participants more likely to rely on the AI prioritising speed and hoping the quality would be acceptable through the AI. The time pressure seemed to cause participants to ignore our subtle friction, meaning most of the mechanisms we had added in an attempt to avoid passive acceptance of AI solutions were ignored due to the time constraint. Interview answers from *Theme 2* support this with a participant stating directly, they might have interacted more with inline feedback if they had not had a sense of time pressure. Most participants, however, often took note of our most intrusive friction: slow accept, as seen in *Theme 1*. This could indicate that the more subtle friction would work better in an environment where the participant does not feel any significant time pressure, and more intrusive friction is required if participants feel under

time pressure. The findings of Seitz et al. [36] and Zhou et al. [41] could also support this notion. Seitz et al. [36] found that under time constraints, the decision-making process may be simplified by focusing on fewer details or prioritising what appears to be the most important information. Zhout et al. [41], using eye-tracking methods, observed that time pressure led to faster decision-making and shorter fixation periods.

As discussed in Section 5.1, there was no significant difference in frustration between groups, but several participants mentioned finding slow accept frustrating. It is possible that adding several intrusive friction types would cause a significant difference in frustration between the non-friction and friction group. A potential benefit of keeping more subtle friction types is low frustration. Therefore, in cases with low time pressure, participants could be more likely to interact with the friction without a significant increase in frustration compared to using more intrusive friction types.

The participant feedback suggests that in high time pressure situations, stronger friction may be necessary to achieve the desired interaction, even if it leads to increased frustration. Conversely, in low time pressure scenarios, more subtle friction might still guide user behaviour effectively without significantly raising frustration levels. In systems where high quality code is paramount, the subtle friction could be beneficial. Slowing down interaction but nurturing deeper engagement and understanding of code for the user without a significant increase in frustration.

5.3 Bias Towards Educational Framing

As highlighted in *Theme 5*, many participants tended to evaluate the friction implementations through an educational lens, despite the experiment not being explicitly framed as a learning exercise. This bias might have emerged due to a combination of factors. First, both participants and facilitators were university students, and the experiment was conducted in an academic environment at the university faculty building. These contextual cues may have implicitly signalled a pedagogical intent, priming participants to view the friction elements as an aid for learning rather than something to be used in a professional context while engaging more thoughtfully with the AI.

Moreover, slow accept and inline feedback may be interpreted as 'training wheels' designed for beginner programmers, as they resemble scaffolding techniques often used in educational tools [16, 20, 23]. These elements, while intended to encourage reflection, can signal a lack of trust in the participant's competence, potentially, leading more experienced programmers to view them as unnecessary or condescending due to using AI as a tool for increasing productivity [34].

This aligns with prior findings from Faber et al. [10], who explains that scaffolding may hinder performance and increase cognitive load, if it is not tailored to the learner's expertise. This supports the expertise reversal effect [18] and underlines the importance of contingent support.

We fear that this educational framing may therefore have inadvertently diminished the perceived relevance or legitimacy of friction in real-world software development. If users assume that friction is meant to teach rather than support, they may have changed how they interact with the system.

5.4 Limitations & Future Work

All participants were recruited through our network and were at the time studying a Master's in Software Engineering at Aalborg University, which also lead to a male-only participant pool. Preferably we would have looked at the effect for professional developers rather than an educational setting, however, our participants and experiment resulted in a middle ground where we do not look at it in an educational setting but a lot of the participants automatically came to think of our extension as an educational tool as seen in *Theme 5*. In addition, as seen in *Theme 3* many participants used more AI as they had not used Python in quite some time so they had to relearn syntax and concepts.

Another limitation was that the participants only spent 25 minutes using the extension. A short experiment may not show the full extent of how the friction affects the interaction and use of AI programming tools. To get results that better reflect the impact of the friction a long term study would be more suitable. It probably did not help that many participants felt a time pressure to complete as many tasks as possible during the experiment, as seen in *Theme 2*, even though they were all informed that they did not have to complete all tasks.

For our experiment it seems that the more subtle implementations of intentional friction may not be intrusive enough to make an impact on all users. In time-sensitive settings, more intrusive types of friction, such as slowing down the acceptance of suggestions, may be more effective. We therefore suggest using intrusive types of frictions for shorter, goal-oriented tasks. At the same time, less intrusive forms of friction may still hold value and could be better suited for long-term use in environments without time pressure. We therefore suggest conducting further long-term studies to explore their potential impact in those contexts.

6 CONCLUSION

In this paper, we have explored the impact of intentionally introducing friction into AI-assisted programming tools, specifically GitHub Copilot, to measure the impact on cognitive load. By deliberately slowing user interaction through mechanisms such as slow acceptance of code suggestions, inline feedback, and highlighting AI-generated code, this research contributes to the broader discourse on friction in human-AI interaction. Our work is among the first empirical studies to investigate the subtler aspects of friction in AI-driven software development environments, highlighting that subtle friction mechanisms may be insufficiently impactful under time constraints. Our thematic analysis provided deeper qualitative insight into the participants' experiences during the experiment. We found that user perception of friction is likely shaped by contextual factors such as time pressure or educational framing, and it might cause users to disregard more subtle friction elements. And when it was noticed, their responses were polarised, some found them frustrating or intrusive, while others recognised their value

and praised them as thoughtful design interventions. Furthermore, some participants tended to prompt first and then later try to understand the AI code if it did not compile. Some participants regretted their usage of AI, as the code became difficult to understand and had errors.

Moving forward, future work should examine long-term interactions with friction-enabled tools in realistic, professional contexts to better understand their sustained impact on developer cognition and code quality. Additionally, exploring more explicitly intervening friction mechanisms is another logical step to exploring the potential of finding a significant difference in cognitive load. By continuing to explore these dimensions, we hope for a future where AI tools not only augment coding efficiency but also fundamentally enhance programmers' critical thinking and decision-making processes.

7 ACKNOWLEDGEMENTS

We thank Niels van Berkel for advising us. We would also like to thank all participants for their help.

REFERENCES

[1] ACM, IEEE-CS, and AAAI. 2024. Computer Science Curricula 2023. https://csed.acm.org/final-report/.

[2] Jared Bauer. 2024. Does GitHub Copilot improve code quality? Here's what the data says. https://github.blog/news-insights/research/does-github-copilot-improve-code-quality-heres-what-the-data-says/

[3] Andrea Benedetti and Michele Mauri. 2023. Design for friction. An inquiry to position friction as a method for reflection in design interventions. *Convergences - Journal of Research and Arts Education* 16, 31 (May 2023). https://doi.org/10.53681/c1514225187514391s.31.139

[4] Zana Buçinca, Maja Barbara Malaya, and Krzysztof Z. Gajos. 2021. To Trust or to Think: Cognitive Forcing Functions Can Reduce Overreliance on AI in AI-assisted Decision-making. *Proc. ACM Hum.-Comput. Interact.* 5, CSCW1, Article 188 (April 2021), 21 pages. https://doi.org/10.1145/3449287

[5] Federico Cabitza, Andrea Campagner, Riccardo Angius, Chiara Natali, and Carlo Reverberi. 2023. AI Shall Have No Dominion: on How to Measure Technology Dominance in AI-supported Human decision-making. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems* (Hamburg, Germany) *(CHI '23)*. Association for Computing Machinery, New York, NY, USA, Article 354, 20 pages. https://doi.org/10.1145/3544548.3581095

[6] Zeya Chen and Ruth Schmidt. 2024. Exploring a Behavioral Model of "Positive Friction" in Human-AI Interaction. In *Design, User Experience, and Usability*, Aaron Marcus, Elizabeth Rosenzweig, and Marcelo M. Soares (Eds.). Springer Nature Switzerland, Cham, 3–22. https://doi.org/10.1007/978-3-031-61353-1_1

[7] Anna L. Cox, Sandy J.J. Gould, Marta E. Cecchinato, Ioanna Iacovides, and Ian Renfree. 2016. Design Frictions for Mindful Interactions: The Case for Microboundaries. In *Proceedings of the 2016 CHI Conference Extended Abstracts on Human Factors in Computing Systems* (San Jose, California, USA) *(CHI EA '16)*. Association for Computing Machinery, New York, NY, USA, 1389–1397. https://doi.org/10.1145/2851581.2892410

[8] Kevin Zheyuan Cui, Mert Demirer, Sonia Jaffe, Leon Musolff, Sida Peng, , and Tobias Salz. 2025. The Effects of Generative AI on High-Skilled Work: Evidence from Three Field Experiments with Software Developers. http://dx.doi.org/10.2139/ssrn.4945566.

[9] Sander de Jong, Ville Paananen, Benjamin Tag, and Niels van Berkel. 2025. Cognitive Forcing for Better Decision-Making: Reducing Overreliance on AI Systems Through Partial Explanations. *Proc. ACM Hum.-Comput. Interact.* 9, 2, Article CSCW048 (May 2025), 30 pages. https://doi.org/10.1145/3710946

[10] Tjitske J E Faber, Mary E W Dankbaar, Walter W van den Broek, Laura J Bruinink, Marije Hogeveen, and Jeroen J G van Merriënboer. 2024. Effects of adaptive scaffolding on performance, cognitive load and engagement in game-based learning: a randomized controlled trial. *BMC Med. Educ.* 24, 1 (Aug. 2024), 943.

[11] Ya Gao, Phillip Coppney, Daniel A. Schocke, Sida Peng, Dan Tetrick, Jeff Wilcox, Erik Polzin, and Lizzie Redford. 2024. Research: Quantifying GitHub Copilot's impact in the enterprise with Accenture. https://github.blog/news-insights/research/research-quantifying-github-copilots-impact-in-the-enterprise-with-accenture/

[12] Muhammad Hamza, Dominik Siemon, Muhammad Azeem Akbar, and Tahsinur Rahman. 2024. Human-AI Collaboration in Software Engineering: Lessons Learned from a Hands-On Workshop. In *Proceedings of the 7th ACM/IEEE International Workshop on Software-Intensive Business* (Lisbon, Portugal) *(IWSiB '24)*. Association for Computing Machinery, New York, NY, USA, 7–14. https://doi.org/10.1145/3643690.3648236

[13] Wei Huang, Jing Sun, and Yang Zhao. 2024. ChatDBG: An AI-Powered Debugging Assistant. *arXiv preprint* (2024). https://arxiv.org/abs/2403.16354

[14] Baskhad Idrisov and Tim Schlippe. 2024. Program Code Generation with Generative AIs. *Algorithms* 17, 2 (2024). https://doi.org/10.3390/a17020062

[15] Martin Jonsson and Jakob Tholander. 2022. Cracking the code: Co-coding with AI in creative programming education. In *Proceedings of the 14th Conference on Creativity and Cognition* (Venice, Italy) *(C&C '22)*. Association for Computing Machinery, New York, NY, USA, 5–14. https://doi.org/10.1145/3527927.3532801

[16] Breanna Jury, Angela Lorusso, Juho Leinonen, Paul Denny, and Andrew Luxton-Reilly. 2024. Evaluating LLM-generated Worked Examples in an Introductory Programming Course. In *Proceedings of the 26th Australasian Computing Education Conference* (Sydney, NSW, Australia) *(ACE '24)*. Association for Computing Machinery, New York, NY, USA, 77–86. https://doi.org/10.1145/3636243.3636252

[17] Daniel Kahneman. 2011. *Thinking, fast and slow.* Farrar, Straus and Giroux, New York.

[18] Slava Kalyuga, Paul Ayres, Paul Chandler, and John Sweller. 2003. The expertise reversal effect. *Educ. Psychol.* 38, 1 (Jan. 2003), 23–31.

[19] Majeed Kazemitabaar, Oliver Huang, Sangho Suh, Austin Z Henley, and Tovi Grossman. 2025. Exploring the Design Space of Cognitive Engagement Techniques with AI-Generated Code for Enhanced Learning. In *Proceedings of the 30th International Conference on Intelligent User Interfaces (IUI '25)*. Association for Computing Machinery, New York, NY, USA, 695–714. https://doi.org/10.1145/3708359.3712104

[20] Majeed Kazemitabaar, Runlong Ye, Xiaoning Wang, Austin Zachary Henley, Paul Denny, Michelle Craig, and Tovi Grossman. 2024. CodeAid: Evaluating a Classroom Deployment of an LLM-based Programming Assistant that Balances Student and Educator Needs. In *Proceedings of the 2024 CHI Conference on Human Factors in Computing Systems* (Honolulu, HI, USA) *(CHI '24)*. Association for Computing Machinery, New York, NY, USA, Article 650, 20 pages. https://doi.org/10.1145/3613904.3642773

[21] Melina Klepsch, Florian Schmitz, and Tina Seufert. 2017. Development and Validation of Two Instruments Measuring Intrinsic, Extraneous, and Germane Cognitive Load. *Frontiers in Psychology* Volume 8 - 2017 (2017). https://doi.org/10.3389/fpsyg.2017.01997

[22] Mohammad Amin Kuhail, Sujith Samuel Mathew, Ashraf Khalil, Jose Berengueres, and Syed Jawad Hussain Shah. 2024. "Will I be replaced?" Assessing ChatGPT's effect on software development and programmer perceptions of AI tools. *Science of Computer Programming* 235 (2024), 103111. https://doi.org/10.1016/j.scico.2024.103111

[23] Mark Liffiton, Brad E Sheese, Jaromir Savelka, and Paul Denny. 2024. CodeHelp: Using Large Language Models with Guardrails for Scalable Support in Programming Classes. In *Proceedings of the 23rd Koli Calling International Conference on Computing Education Research* (Koli, Finland) *(Koli Calling '23)*. Association for Computing Machinery, New York, NY, USA, Article 8, 11 pages. https://doi.org/10.1145/3631802.3631830

[24] Yue Liu, Thanh Le-Cong, Ratnadira Widyasari, Chakkrit Tantithamthavorn, Li Li, Xuan-Bach D. Le, and David Lo. 2024. Refining ChatGPT-Generated Code: Characterizing and Mitigating Code Quality Issues. *ACM Trans. Softw. Eng. Methodol.* 33, 5, Article 116 (June 2024), 26 pages. https://doi.org/10.1145/3643674

[25] Thomas Mejtoft, Sarah Hale, and Ulrik Söderström. 2019. Design Friction. In *Proceedings of the 31st European Conference on Cognitive Ergonomics* (BELFAST, United Kingdom) *(ECCE '19)*. Association for Computing Machinery, New York, NY, USA, 41–44. https://doi.org/10.1145/3335082.3335106

[26] Arghavan Moradi Dakhel, Vahid Majdinasab, Amin Nikanjam, Foutse Khomh, Michel C. Desmarais, and Zhen Ming (Jack) Jiang. 2023. GitHub Copilot AI pair programmer: Asset or Liability? *Journal of Systems and Software* 203 (2023), 111734. https://doi.org/10.1016/j.jss.2023.111734

[27] Satya Nadella, Amy Hood, and Brett Iverson. 2024. Microsoft Fiscal Year 2024 Second Quarter Earnings Conference Call. https://www.microsoft.com/en-us/investor/events/fy-2024/earnings-fy-2024-q2

[28] Mohammad Naiseh, Reem S. Al-Mansoori, Dena Al-Thani, Nan Jiang, and Raian Ali. 2021. Nudging through Friction: An Approach for Calibrating Trust in Explainable AI. In *2021 8th International Conference on Behavioral and Social Computing (BESC)*. IEEE, 1–5. https://doi.org/10.1109/BESC53957.2021.9635271

[29] Nathalia Nascimento, Paulo Alencar, and Donald Cowan. 2023. Artificial Intelligence vs. Software Engineers: An Empirical Study on Performance and Efficiency using ChatGPT. In *Proceedings of the 33rd Annual International Conference on Computer Science and Software Engineering* (Las Vegas, NV, USA) *(CASCON '23)*. IBM Corp., USA, 24–33. https://dl.acm.org/doi/10.5555/3615924.3615927

[30] Nhan Nguyen and Sarah Nadi. 2022. An empirical evaluation of GitHub copilot's code suggestions. In *Proceedings of the 19th International Conference on Mining Software Repositories* (Pittsburgh, Pennsylvania) *(MSR '22)*. Association for Computing Machinery, New York, NY, USA, 1–5. https://doi.org/10.1145/3524842.3528470

[31] Ruchi Panchanadikar and Guo Freeman. 2024. "I'm a Solo Developer but AI is My New Ill-Informed Co-Worker": Envisioning and Designing Generative AI to Support Indie Game Development. *Proc. ACM Hum.-Comput. Interact.* 8, CHI PLAY, Article 317 (Oct. 2024), 26 pages. https://doi.org/10.1145/3677082

[32] Sida Peng, Eirini Kalliamvakou, Peter Cihon, and Mert Demirer. 2023. The Impact of AI on Developer Productivity: Evidence from GitHub Copilot. arXiv:2302.06590 [cs.SE] https://arxiv.org/abs/2302.06590

[33] James Prather, Brent N. Reeves, Paul Denny, Brett A. Becker, Juho Leinonen, Andrew Luxton-Reilly, Garrett Powell, James Finnie-Ansley, and Eddie Antonio Santos. 2023. "It's Weird That it Knows What I Want": Usability and Interactions with Copilot for Novice Programmers. *ACM Trans. Comput.-Hum. Interact.* 31, 1, Article 4 (Nov. 2023), 31 pages. https://doi.org/10.1145/3617367

[34] James Prather, Brent N Reeves, Juho Leinonen, Stephen MacNeil, Arisoa S Randrianasolo, Brett A. Becker, Bailey Kimmel, Jared Wright, and Ben Briggs. 2024. The Widening Gap: The Benefits and Harms of Generative AI for Novice Programmers. In *Proceedings of the 2024 ACM Conference on International Computing Education Research - Volume 1* (Melbourne, VIC, Australia) *(ICER '24)*. Association for Computing Machinery, New York, NY, USA, 469–486. https://doi.org/10.1145/3632620.3671116

[35] Daniel Russo. 2024. Navigating the Complexity of Generative AI Adoption in Software Engineering. *ACM Trans. Softw. Eng. Methodol.* 33, 5, Article 135 (June 2024), 50 pages. https://doi.org/10.1145/3652154

[36] Florian I. Seitz, Bettina von Helversen, Rebecca Albrecht, Jörg Rieskamp, and Jana B. Jarecki. 2023. Testing three coping strategies for time pressure in categorizations and similarity judgments. *Cognition* 233 (2023), 105358. https://doi.org/10.1016/j.cognition.2022.105358

[37] Inbal Shabi and GitHub Staff. 2024. Survey reveals AI's impact on the developer experience. https://github.blog/news-insights/research/survey-reveals-ais-impact-on-the-developer-experience/.

[38] Nigar M. Shafiq Surameery and Mohammed Y. Shakor. 2023. Use Chat GPT to Solve Programming Bugs. *International Journal of Information Technology & Computer Engineering* 3, 01 (Jan. 2023), 17–22. https://doi.org/10.55529/ijitc.31.17.22

[39] John Sweller. 1988-04. Cognitive Load During Problem Solving: Effects on Learning. *Cognitive science.* 12, 2 (1988-04). https://doi.org/10.1207/s15516709cog1202_4

[40] John Sweller, Jeroen J. G. van Merrienboer, and Fred G. W. C. Paas. 1998. Cognitive Architecture and Instructional Design. https://doi.org/10.1023/A:1022193728205. , 46 pages.

[41] Yan-Bang Zhou, Shun-Jie Ruan, Kun Zhang, Qing Bao, and Hong-Zhi Liu. 2024. Time pressure effects on decision-making in intertemporal loss scenarios: an eye-tracking study. *Frontiers in Psychology* Volume 15 - 2024 (2024). https://doi.org/10.3389/fpsyg.2024.1451674

[42] Rina Zviel-Girshin. 2024. The Good and Bad of AI Tools in Novice Programming Education. *Education Sciences* 14, 10 (2024). https://doi.org/10.3390/educsci14101089

[43] Mert İnan, Anthony Sicilia, Suvodip Dey, Vardhan Dongre, Tejas Srinivasan, Jesse Thomason, Gökhan Tür, Dilek Hakkani-Tür, and Malihe Alikhani. 2025. Better Slow than Sorry: Introducing Positive Friction for Reliable Dialogue Systems. arXiv:2501.17348 [cs.CL] https://arxiv.org/abs/2501.17348

Appendix A EXPERIMENT SCRIPT

Manuskript til forsøget

Før forsøget

- ☐ “Velkommen og tak fordi du gad at deltage i vores eksperiment”
- ☐ “Det kommer til at virke lidt unaturligt, men under hele forsøget følger vi et manuskript. Du er stadig velkommen til at stille spørgsmål undervejs, hvis du er i tvivl om noget”
- ☐ “Tag venligst plads her” ***vis dem frem til pladsen***
- ☐ “Før vi går i gang med forsøget, bedes du læse og underskrive en samtykkeerklæring” ***giv dem samtykkeerklæringen***
- ☐ “Du bedes også svare på dette spørgeskema om din programmeringserfaring" ***giv dem spørgeskemaet***
- ☐ “Du skal snart til at kode et enkelt program i Python med hjælp fra GitHub Copilot. Det kommer til at foregå, således, at du får udleveret delopgaver på et stykke papir, hvor du har 25 minutter til at nå så langt du kan. Det er ikke vigtigt, at du når at blive færdig med alle opgaver, og du kan altid vælge at stoppe undervejs.”
- ☐ “Efter tiden er gået eller du ikke vil fortsætte vil du få endnu et spørgeskema, som bliver efterfulgt af et interview”
- ☐ “Programmeringsdelen kommer til at foregå i Visual Studio Code. Når du skal køre koden, bliver du nødt til at gøre det i terminalen.”
- ☐ **(FRICTION ONLY)** “Det er meget vigtigt at der står @tutor først i hver prompt du skriver til Copilot.”
- ☐ “Her kan du se timeren. Når du er klar kan du starte den, hvorefter du vil få opgaverne udleveret. Du er selvfølgelig velkommen til at tjekke timeren løbende” ***vis timeren,***
- ☐ “Værsgo” ***giv opgave ved timer start***

Efter forsøget

- ☐ “Forsøget er nu færdigt. Du bedes nu svare på dette spørgeskema” ***udlever spørgeskema***
- ☐ “Vi vil nu gå igang med et lille interview om din oplevelse da du programmerede”
- ☐ “Vi kommer til at optage lyd til interviewet for at gøre dataindsamlingen nemmere” ***start lydoptagelse på bærbaren og derefter interviewet***
- ☐ “Så er der ikke flere spørgsmål. Ønsker du at få at vide præcis hvad vi undersøgte?”
- ☐ “Det kunne være fedt, hvis du ikke snakker til andre forsøgspersoner du kender om eksperimentet, indtil de også har været igennem det”
- ☐ “Det var sådan set alt. Mange tak for din deltagelse”
- ☐ ***tilbyd snack og få dem til at smutte***

Appendix B EXPERIMENT SURVEY

Pre-tasks questionnaire/survey

1. Age: ___

2. How many years of experience do you have with programming: ___

3. In what contexts have you programmed? (set one or more Xs):
   - ☐ Academically
   - ☐ Hobby projects
   - ☐ Work
   - ☐ Other: (please specify)

4. How many years of experience do you have with python: ___

5. In what contexts have you used python? (set multiple X):
   - ☐ Academically
   - ☐ Hobby projects
   - ☐ Work
   - ☐ Other: (please specify)

6. When working on coding tasks, how often do you use AI tools (e.g., ChatGPT, GitHub Copilot)?
   - ☐ Never
   - ☐ Rarely
   - ☐ Sometimes
   - ☐ Often
   - ☐ Always

7. For what do you usually use AI tools while coding? (set one or more X)
   - ☐ I never use AI tools while coding.
   - ☐ Generating entire code snippets
   - ☐ Debugging
   - ☐ Code completion/autocompletion
   - ☐ Learning new concepts or syntax
   - ☐ Other: (please specify)

Appendix C EXPERIMENT TASKS

Objective:

Create a program that analyzes student grade data, performs various calculations, and generates a report in python.

Subtasks:

Load data

1. Extract the CSV file data into appropriate data strictures
2. Ensure that scores are valid (between 0-100). If the score is invalid ignore this entry for the overall data extraction. Potentially store the incorrect scores separately from the correct scores.
3. Error handling for missing data types. Remove the entry unless the key information is still present. For example, the name is missing but everything else is still present.

Grade assignments

4. Calculate weighted final grade with:
    - Quizzes: 15%
    - Assignments: 20%
    - Midterm: 25%
    - Final: 40%
5. Assign grades based on this scale:
    - 12: 90-100
    - 10: 80-89
    - 7: 70-79
    - 4: 60-69
    - 02: 50-59
    - 00: 30-49
    - -03: 0-29

Grade average

6. Calculate the mean, median and standard deviation for each students performance in each subject. For example, their mean grade in math is based on quizzes, assignments, midterm and final.
7. Calculate the mean, median and standard deviation for all students' overall performance in each subject. For example: the mean grade in math for all students could be 8.7
8. Identify the highest-performing and lowest-performing student in each subject
9. Identify the subject students overall had the best performance in and the subject students overall had the worst performance in.

Progress Tracking

10. For each student, calculate improvement/decline between assignments and quizzes
11. Identify consistent performers vs. improving/declining students
12. Flag students who might need additional support due to consistently lower scoring or dropping scores over time.

Multiple Report Generation

13. Create an individual student report showing detailed performance
14. Generate a detailed report for performance in each subject
15. Produce a report highlighting at-risk students due to their poor performance

Command-line Interface

16. Implement a simple menu-driven interface for different analysis options
17. Allow filtering students by various criteria (grade range, improvement status)
18. Support exporting reports in different formats (text, CSV)

Appendix D CLTFC QUESTIONNAIRE

	(Strongly disagree) 1	2	3	4	5	6	(Strongly agree) 7
While solving the tasks, many things needed to be kept in mind simultaneously.	○	○	○	○	○	○	○
The tasks were very complex.	○	○	○	○	○	○	○
I made an effort, not only to understand several details, but to understand the overall context.	○	○	○	○	○	○	○
My goal while dealing with the tasks was to understand everything correctly.	○	○	○	○	○	○	○
During the tasks, it was exhausting to find the important information.	○	○	○	○	○	○	○
The design of the tasks was very inconvenient for learning.	○	○	○	○	○	○	○
During the tasks, it was difficult to recognize and link the crucial information.	○	○	○	○	○	○	○
I am confident in my solutions to the solved tasks.	○	○	○	○	○	○	○
Solving the tasks was frustrating.	○	○	○	○	○	○	○

Appendix E EXPERIMENT INTERVIEW QUESTIONS

BED OM AT UDDYBE HVIS DE SVARER JA/NEJ

Generelle spørgsmål:

1. Hvad synes du om brugen af AI værktøjer til programmering?
2. Føler du, at du prøvede at forstå koden som AI'en producerede, inden du brugte den?
3. Syntes du at AI'en hjalp dig undervejs?
    a. Uddyb, hvorfor/hvordan (ikke)
4. Følte du, at AI'en fik dig til at programmere hurtigere?
    a. Uddyb, hvorfor/hvordan (ikke)
5. Brugte du AI mere end du normalt ville have gjort?
    a. Uddyb, hvorfor/hvordan (ikke)

Kun til deltagere med friktion:

1. Hvad syntes du om at AI'en gav dig inline feedback til din kode?
2. Hvad syntes du om at du ikke kunne acceptere copilot suggestions uden at skulle skrive det ud selv?
3. Hvad syntes du om at copy-pasted kode fra AI chatten blev markeret med highlighting?
4. Fik friktionen dig til at ændre hvordan du løste opgaverne?
    a. Uddyb, hvorfor/hvordan (ikke)
5. Følte du, at friktionen hjalp dig med at forstå koden, eller var det mest af alt en forhindring?
    a. Uddyb, hvorfor/hvordan

Kun til deltagere UDEN friktion:

1. Tror du, at du har støttet dig for meget op af AI'en?
2. Hvad synes du om at implementere friktion, hvor vi forstyrrer brugeren undervejs for at få dem til at tænke mere over AIens svar?