Putting Brains Back in the Loop: A Mastery-Aware Pipeline for LLM-Assisted Software Development

Vicente Corrêa da Silva Neto
Pontifical Catholic University of Rio de Janeiro (PUC-Rio)
Rio de Janeiro, Brazil
vneto@tecgraf.puc-rio.br

Alessandro Garcia
Pontifical Catholic University of Rio de Janeiro (PUC-Rio)
Rio de Janeiro, Brazil
afgarcia@inf.puc-rio.br

Daniel Coutinho
Pontifical Catholic University of Rio de Janeiro (PUC-Rio)
Rio de Janeiro, Brazil
dcoutinho@inf.puc-rio.br

Anderson José Silva de Oliveira
SERPRO
Brazil
anderson.jose.so@gmail.com

João Correia
Pontifical Catholic University of Rio de Janeiro (PUC-Rio)
Rio de Janeiro, Brazil
jcorreia@inf.puc-rio.br

Abstract

Large Language Model (LLM)-based coding assistants are transforming software development by increasing short-term productivity, but their impact on developer understanding remains unclear. Emerging evidence suggests that intensive reliance on these tools may lead to *cognitive bypass*, where developers generate and accept assistants' correct solutions without fully internalizing their logic, resulting in accumulated *cognitive debt*. Existing work does not empirically assess the extent of cognitive debt in typical programming tasks and how it can be reduced with structured assistant-human interventions. In this paper, we propose the *Mastery-Aware Pipeline*, a workflow that introduces structured *Mastery Checks*, including pre-implementation specification and post-implementation Socratic review, to promote active reasoning in AI-assisted development. We evaluate this approach through a quasi-experiment with 19 participants performing programming tasks with and without the pipeline. Our results show that structured checks are associated with higher developer understanding: average mastery scores were higher with the pipeline (73) than without it (62), and the proportion of correct solution explanations was nearly 70% with the pipeline versus 32% without it. We also observed that unconstrained AI usage led to more variable outcomes, reduced debugging confidence, and a sharper decline in reflective practices. These findings suggest that workflow design, such as the *Mastery-Aware Pipeline*, plays a critical role in mediating the cognitive effects of LLM-assisted development. Lightweight structured interventions can help preserve both productivity and developer understanding.

Keywords

Large Language Models, Cognitive Debt, Software Engineering, Human-in-the-Loop, Developer Productivity

1 Introduction

Software developers are increasingly integrating Large Language Models (LLMs) into their software development workflows [17]. This phenomenon is changing how developers write, review, and reason about code [25]. Tools such as GitHub Copilot[1], Claude Code[2], and OpenAI Codex[3] are now deeply embedded in Integrated Development Environments (IDEs), enabling automated code generation, completion, and explanation. Empirical studies report substantial productivity gains associated with these tools [22], particularly by reducing time spent on routine programming tasks.

Despite these benefits, emerging evidence suggests that intensive reliance on LLM-based assistants may alter the cognitive processes involved in software development [18, 19]. While traditional programming requires developers to actively construct and refine mental models of system behavior, LLM-assisted workflows can shift this process toward evaluating generated outputs. Preliminary studies and anecdotal reports indicate that developers may increasingly delegate problem decomposition and solution synthesis to AI systems, engaging primarily in verification rather than construction, the so-called Cognitive Bypass [18, 19]. This phenomenon is a development pattern in which developers rely on AI-generated solutions without fully internalizing the underlying logic, thereby bypassing the cognitive effort required to build robust mental models [18, 19]. In this mode of interaction, developers risk adopting a passive role, approving generated code without critically engaging with its structure or implications.

This risk is further exacerbated by two complementary factors identified in recent literature. First, the capabilities of LLMs have been described as a *Jagged Technological Frontier* [7], in which models perform unevenly across tasks: problems that appear complex to humans may be handled correctly, while seemingly simple tasks may fall outside the model's competence, often resulting in confident but incorrect outputs. Second, this uneven capability interacts with *Automation Bias*, a well-documented tendency for humans to favor suggestions from automated systems, reducing their own critical engagement [26]. Empirical evidence suggests that developers assisted by AI systems may over-trust generated code, leading to poorer security outcomes compared to unaided development [24]. Together, these factors amplify the risks of Cognitive Bypass, as developers may both defer to and insufficiently scrutinize AI-generated solutions.

[1] https://github.com/features/copilot

[2] https://claude.com/product/claude-code

[3] https://openai.com/codex/

This phenomenon is closely related to the notion of *Cognitive Debt* [30], which refers to the accumulation of gaps in understanding resulting from reduced active engagement with problem-solving tasks [19]. Over time, such gaps may impair a developer's ability to assess code quality, reason about system behavior, and maintain complex codebases. In extreme cases, this may lead to software artifacts that are functionally correct but poorly understood by their contributors, increasing long-term maintenance risks [19]. This type of approach, which maximizes short-term productivity at the expense of cognitive engagement may introduce trade-offs that are not yet fully understood.

In this paper, we argue for a complementary perspective on AI-assisted development, one that preserves the benefits of automation while actively supporting developer learning and understanding. We called it the ***Put Brains Back*** approach. Rather than constraining the use of LLMs, we propose mechanisms to re-engage developers in the reasoning process. Namely, we introduce the concept of *Mastery Checks*: structured interventions embedded within the development workflow that prompt developers to articulate, justify, or adapt generated solutions.

First, developers are required to produce a partial, human-authored specification that outlines the problem, constraints, and intended approach, ensuring initial engagement with the task before any AI-generated solution is considered. Second, after interacting with AI-generated code, developers must demonstrate proof-of-mastery by explaining and critically reflecting on the solution, including its rationale, limitations, and potential edge cases. This second step is achieved through a *Socratic review*, a process in which a tutor (in this case, an LLM) challenges the developer's knowledge through a series of questions designed to stimulate the use of critical thinking [11, 23]. This process is designed to use AI agents to assist, but not replace, the developer's reasoning, enforcing active cognitive participation throughout the development lifecycle.

To evaluate this approach, we conducted a quasi-experiment with 19 participants performing two programming tasks under different conditions: one governed by the proposed Mastery-Aware pipeline and one using an unconstrained AI-assisted workflow. We assessed developer understanding through a composite Mastery Score and complementary qualitative indicators derived from a Socratic review. The results show that, on average, participants achieved higher mastery scores under the structured condition (72.9 vs. 62.2), with a decline observed in the unconstrained condition (15/19 participants). Additionally, the proportion of participants able to correctly explain their solutions dropped from 68.4% to 31.6% without the pipeline, accompanied by reduced debugging confidence and a sharp decrease in reflective documentation practices. These findings suggest that structured interventions are associated with stronger and more consistent evidence of developer understanding, while unconstrained AI usage may lead to more variable outcomes and reduced internalization of generated solutions.

By framing AI assistance not only as a productivity tool but also as a pedagogical instrument, this study aims to advance the design of development workflows that balance efficiency with sustained cognitive engagement. In this context, this study makes four main contributions. First, we provide an empirical characterization of cognitive debt in AI-assisted development, showing that differences in developer understanding can be systematically observed through a combination of quantitative measures (Mastery Score) and qualitative assessments (Socratic evaluation), revealing gaps that are not detectable through functional correctness alone. Second, we present evidence that structured desirable friction is associated with improved developer understanding, as workflows incorporating specification and proof-of-mastery steps are linked to higher mastery scores, better explanation ability, and more consistent reflective practices. Third, we introduce the Mastery-Aware Pipeline, a practical workflow for AI-assisted development that integrates formal specification and Socratic review as lightweight interventions to promote active reasoning without restricting the use of AI tools. Finally, we provide insights into the relationship between AI usage and developer cognition, suggesting that outcomes are influenced more by how developers engage with AI than by experience level or model capability, thereby highlighting the role of workflow design in mediating cognitive effects.

2 Background and Related Work

In this section, we discuss key concepts in our study: how Socratic approaches are applied in the context of LLM use, the cognitive debt on coding tasks, and the use of specification-based agents.

2.1 Socratic Review

Socratic questioning is a process of guided discovery that encourages critical thinking through iterative question-answer dialogue [11, 23]. Studies have explored Socratic-style LLM interaction to complement direct AI answers, prompting users to reason about proposed changes rather than passively accepting them [1, 21]. In teaching scenarios, Ding *et al.* [8] propose SocraticLLM, reframing LLMs from problem solvers into interactive tutors that scaffold reasoning via dialogue. In software development, Liding *et al.* [20] show Socratic hints outperform targeted hints for long-term debugging outcomes, and Della Porta *et al.* [6] demonstrate that structured self-interrogation (MIND) improves LLM reasoning across code generation and repair tasks.

Current studies use the Socratic approach to complement LLMs during task execution; none focuses on post-implementation review of the developer's own understanding. We expand this with a *Socratic Review* in which the LLM agent evaluates both code quality and the knowledge acquired by the developer during the task.

2.2 Cognitive Debt

Cognitive debt accumulates when developers delegate reasoning to AI without fully internalizing the generated results [2, 19, 30]. Kosmyna *et al.* [19] show through a four-month EEG and NLP study ($n = 54$) that ChatGPT users exhibit the lowest neural connectivity, weakest memory recall, and least perceived authorship, raising concerns that productivity gains come at the cost of long-term cognitive development. Complementary systematic reviews show that cognitive load measurement in AI-assisted development is evolving toward multimodal sensing [14, 15, 32].

Our study focuses on cognitive debt in coding tasks, using Socratic reviews to mitigate it and measuring the effect immediately after developers accept AI-generated solutions.

2.3 Specification-based Agents

Spec-Driven Design (SDD) grounds AI agents in explicit, structured specifications rather than underspecified natural-language prompts [4]. Rosa *et al.* [27] propose CURRANTE, a VS Code extension structuring LLM-assisted development into specification, test generation, and function generation stages, showing that structured intent expression improves correctness and efficiency. Broader surveys and human-in-the-loop frameworks confirm the potential of multi-agent pipelines for software engineering tasks [16, 33, 37].

Unlike these studies, we use specification-based agents to guide both the LLM and the developer throughout the task, examining how cognitive debt is reduced or amplified depending on engagement strategy. All programming task instructions are written by humans, and the pipeline combines formal specification with post-implementation Socratic review.

3 The Mastery-Aware Pipeline

In this section we propose the **Mastery-Aware** pipeline, a workflow that governs interaction with AI code assistants, re-establishing the developer role as the active software designer by intentionally introducing structured friction or "desirable difficulty". This concept is presented by Bjork's psychological theory, which states that the effort required to synthesize information enhances knowledge retention through the following affirmations: "manipulations that appear to introduce difficulties for the learner during training can enhance post-training performance" and "the act of retrieving information is itself a potent learning event" [3].

These difficulties are directly related to the framework process, since the developer is required to do more work by formally specifying the programming task requirements through structured artifacts, contrasting with the usual "vibe coding" workflow, where the developer asks the code assistant in a chat session to perform the tasks based on vague or superficial requests. The proposed pipeline is structured around two distinct stages: **Formal Specification** and **Proof of Mastery**, illustrated in Figure 1 and detailed as following.

3.1 Stage 1: Formal Specification

This stage forces the developer to reflect upon his intent and formally specify the task before the code assistant is allowed to start coding the solution. This rule not only introduces the desirable friction that builds the mental model, but also provides the agent with better specifications through the use of the **`TODO.md`** artifact than it would receive during regular LLM chat sessions. The file is structured so that the developer needs to state the problem and articulate the requirements, define the solution steps, and the success criteria before a single line of code is generated.

The AI agent is strictly denied permission to create or update this file. Thus, this restriction ensures the developer experiences the desirable friction of thinking and carefully defining the software design while building the mental model. This stage governance is enforced via **state locking**. If `TODO.md` is empty, missing, or not compatible with the user prompt, the agent refuses to code. This prevents the workflow from proceeding until the human software designer has really thought about the problem, at least on a reasonable degree, instead of simply outsourcing the implementation.

3.2 Stage 2: Proof of Mastery

This stage occurs post-implementation. Its goal is to verify whether the developer has internalized the logic generated by the AI. This stage preludes the Pull Request operation with an active defense of the solution and improves it with historical **"Proof of Mastery"**. Three types of artifacts, which as described below, are used to bridge the gap between AI generation and human comprehension:

`REPORT.md` (AI-Generated): When completing the task, the agent generates a summary of the changes with its implementation applied. Complementing the versioning diff, this serves as an optimized reference for the developer to understand which artifacts she should inspect to understand the modifications made by the code assistant.

`REACTO.md` (Human-Generated): A developer should fill this artifact according to the REACTO [28] technique, that we adapted from coding interviews, obeying the structured description in Table 1.

`SOCRATIC_REVIEW` (AI-Generated after Human vs AI debating): This artifact is generated and derived by the coding assistant from a Socratic debate between the developer and the AI agent itself. The AI agent is instructed to base the debate on the other previously generated artifacts and also on the modifications available through the versioning diff of the project. See Section 3.3 for more details.

3.3 The Socratic Copilot Reviewer

The Socratic Copilot Reviewer is a persona adopted by the AI agent through custom and agentic instructions [12, 13], invoked whenever the developer is satisfied and ready to submit their contribution for further review and integration. Because coding assistants possess the context-awareness to read documentation artifacts, analyze project modifications, and verify automated tests, they provide an ideal environment for a deep, interactive discussion about both high-level architectural decisions and low-level code modifications.

To rigorously assess the developer comprehension and competence, the agent engages them in a Socratic reasoning dialectic method proven to expose hidden assumptions, clarify constraints, and test the validity of intermediate conclusions [6]. Due to the confrontational nature of the Socratic reviewer, the developer needs to understand what is being submitted, reflect on the quality of the submission, and provide historical evidence of his competence via the versioned `SOCRATIC_REVIEW.md` artifact in order to have his contribution accepted by a human without a negative mastery history committed.

3.4 Self-development: The BrainsBack CLI Tool

To validate the feasibility of the *Mastery-Aware* Pipeline, we have implemented the proposed workflow in the development of the *BrainsBack* CLI tool. The project is open-source, available on GitHub [9] and helps applying the proposed workflow in virtually any project, including itself. After the initial boilerplate, all software increments of the tool utilized its own features to ensure the quality and adherence to the proposal. This self-development process recruited *VSCode's GitHub Copilot* as the actual coder of the tool, chosen because it was the precursor of code assistants IDEs and also part of the most popular editor, while the human developer retained the

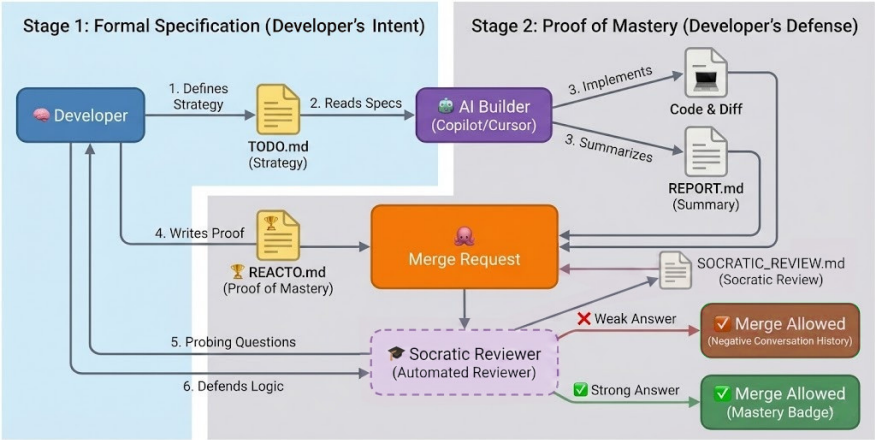

Figure 1: The Mastery-Aware Pipeline Architecture

Table 1: REACTO Technique and the role of each phase in the Mastery-Aware Pipeline

Phase	Description
Repeat	Restate the user-facing problem. This forces the developer to articulate the problem in their own words instead of passively adopting the AI's framing, confirming that human and assistant share the same target before any solution is defended.
Examples	Provide concrete inputs and expected outputs that demonstrate the fix, including edge cases. This grounds the AI-generated changes in observable behavior and exposes whether the developer actually understands what the code does, rather than trusting plausible-looking output.
Approach	Describe the implementation strategy conceptually. This requires the developer to reconstruct the design at an abstract level, distinguishing genuine comprehension of the strategy from mere surface familiarity with the generated code.
Code Rationale	Pinpoint the most critical code changes and justify their design. This shifts the focus from *what* was changed (already visible in the diff) to *why* it was changed, anchoring the proof of mastery in design intent rather than syntactic recognition.
Test Traceability	Explain how the solution was validated, linking test sources to the examples provided or describing which manual tests were performed. This couples claims of correctness to verifiable evidence and counteracts the well-known tendency of coding assistants to over-assert success.
Optimize	Address Big(O) complexity, trade-offs and opportunities for improvement. This pushes reflection beyond "it works", surfacing trade-offs and future improvements that the AI rarely raises spontaneously and signaling a deeper, ownership-level engagement with the contribution.

role of a high-level software designer, specifying, reviewing, and adjusting the product as new requirements were identified.

Illustrative Instance of a contribution and its generated artifacts. Here we describe a real scenario faced in the development of the Self-developed BrainsBack CLI tool [9] in one of the software increments meta-developed with the tool. Trying to run the CLI tool from a global installation would result in a problem resolving dependencies unless the dependencies were locally installed in the working directory. In this particular case, we knew the symptom, but had no idea what the cause was until the investigative and development process evolved with the support of the AI coding assistant. The process took the following steps and generated all the supra-cited artifacts:

Stage 1 - Specification. The developer specified the high-level problem in `TODO.md`, focusing on telling the bug symptoms and asking the Code assistant to investigate and fix the issue and describing the definition of done / success looks like condition (Listing 1).

```
# Strategic Blueprint
## The Problem
The `brainsback` CLI crashes when run after being installed globally via `npm
      install -g .`.
## Steps
- [ ] Investigate the cause of the crash.
- [ ] Fix the issue.
## Success Looks Like
- [ ] All `brainsback` commands run without error after a global installation.
```

Listing 1: TODO.md

With the `TODO.md` artifact in place, the AI agent is allowed to proceed upon user request. This request is made through the usual Copilot chat interface and triggers a collaborative interaction between the developer and the agent in a Pair-Programming-like session: The human developer and the AI agent investigate, analyze command outputs, discuss, and implement code changes in a very similar way to what regular co-workers would do. This iterative dialogue continues until the problem is solved and all tests pass. The process culminates in two key outputs: the final code diff and the agent-generated `REPORT.md` (Listing 2), which summarizes the changes and their rationale.

```
# Implementation Report
## Snapshot
- **Change**: Fix `brainsback` global install failing to resolve dependencies
- **Status**: Complete
## The Changes
- `lib/commands/init.js` - Dynamic path resolution with dev/production fallback
- `bin/brainsback.js` - Entry point updated to reference the dist bundle
- `package.json` - `bin` now points to `dist/brainsback.js`; esbuild script
      added
## Testing Strategy
- `npm run build` runs without error and generates `dist/brainsback.js`
- Installed globally; `brainsback init` and `brainsback status` work
  in an empty directory
```

Listing 2: REPORT.md

Stage 2 - Mastery. In the second stage, the developer review the code assistant modifications, the generated `REPORT.md` artifact and try to prove his comprehension through the `REACTO.md` (Listing 3) artifact. After filling the `REACTO.md` artifact, the developer can ask the coding assistant to assess his described understanding of the solution. The developer and the agent continue the conversation until they are confident that the artifact indeed represents the work accomplished.

```
# Proof of Mastery (REACTO)
## R - The Problem
The CLI tool crashed when installed globally (`npm install -g`) because...
## E - Examples
- Input: User runs `npm install -g .` then `brainsback init` in a new folder.
  Output: command executes successfully using bundled dependencies; no "module
      not found" errors.
## A - Approach
Switch from distributing raw source files to distributing a single bundled
      executable.
1. Use `esbuild` to compile the app and all dependencies into `dist/brainsback.
      js`.
2. Configure `package.json` to publish the bundle instead of the `lib/` folder.
## C - Code
- Dynamic Path Resolution with Fallback: In `lib/commands/init.js`, we couldn't
      rely on a static relative path because the directory depth changes
      between source (`lib/commands/init.js`) and bundle (`dist/brainsback.js`).
       We implemented a check using `fs.existsSync` to verify
## T - Tests
- Build Verification: `npm run build` runs without error and generates `dist/
      brainsback.js`.
- Artifact Inspection: Verified `dist/brainsback.js` starts with exactly one
      Shebang line.
- Manual Integration: Installed the package globally from the local source and
      verified `brainsback` commands function in an empty directory.
## O - Optimization
Doesn't really apply in this scenario
```

Listing 3: REACTO.md

When both are satisfied with the solution, the developer can finally invite the Socratic Reviewer persona, which is also governed by GitHub Copilot's Custom Agents [13] and Skills [12], directly within the chat session to conduct a final *Proof-Of-Mastery* discussion. The agent scans `REACTO.md` and `REPORT.md` artifacts, reads the code diff, checks if `REACTO.md` is sufficiently complete, and then proceeds to a Socratic [6], not pre-determined, debate until the final approval is achieved and the last artifact `SOCRATIC_REVIEW.md` (Listing 4) is generated. At this point, all the modifications can be submitted for a PULL REQUEST for a hopefully human review and acceptance.

4 Study Design

We conducted a quasi-experiment [29] to evaluate the impact of the Mastery-Aware Pipeline on cognitive debt. In this experiment, we

```
## Session
- **Date**: 2026-04-20
- **Change**: Fix brainsback global install dependency resolution
- **Verdict**: MASTERY PROVEN

## Exchange

### Q1
**Question**: You use `fs.existsSync` to pick the template path at runtime.
What happens if neither the dev nor the production path exists, and why is
that acceptable?
...
### Q2
**Question**: Why switch from ESM dynamic `import()` to CommonJS `require`?
What would have broken if you had kept dynamic imports?

**Answer**: esbuild cannot statically analyse dynamic `import()` calls, so
those modules would remain unresolved in the single-file bundle. Switching
to `require` lets esbuild inline every dependency, making the global
executable fully self-contained.
```

Listing 4: SOCRATIC_REVIEW.md

submit the developers to two distinct tasks in a Tic-Tac-Toe game: the 1st task using the pipeline and the 2nd task without using it.

4.1 Research Questions

With this quasi-experiment, we aimed to understand how the usage of AI code assistants can reflect on the accumulation of cognitive debt and to what extent. Mostly important, we analyze whether we can mitigate or diminish the extent of this phenomenon by reintroducing desirable humans' friction. With that goals in mind, we defined the following research questions.

RQ1. *Can cognitive debt be systematically detected and measured?* To what extent can the existence of cognitive debt be systematically detected and quantified within an AI-assisted development workflow? To answer **RQ1**, we utilized a Socratic debate phase (Section 3.3) that the participants should participate in after completing all the tasks to prove and/or expand his mastery of the contribution.

RQ2. *Does structured desirable friction mitigate cognitive debt?* What is the effect of introducing desirable friction, such as structured constraints and Socratic interactions, as a mechanism to mitigate cognitive debt? To answer **RQ2**, we compared the performance of the participants on the Socratic debate in both tasks and calculated the differences in Task 1 (with the mastery pipeline) and Task 2 (without the pipeline). The evaluated metrics on this activity were *Module Explanation*, *Socratic Debate* and *Design Justification*. These three basic metrics were chosen as they are important to characterize the extent of the cognitive engagement. While critical thinking is captured by the socratic debate, the explanation of the core modules involved in the programming tasks and design justification are additional, critical parts of the cognitive process that implies basic knowledge retention.

4.2 Recruiting Participants

To select participants, we recruited developers with professional-level experience ranging from 2 to 10 years in software development, as well as diverse academic backgrounds. The sample included participants from undergraduate students in the third year of Computer Science or Computer Engineering to more experienced developers

including those with MSc, PhD, and postdoctoral degrees. All the selected undergraduate students participate in software projects and are already experienced with the use of AI assistants, i.e., they previously enrolled in programming tasks with support of these assistants. These diverse backgrounds were convenient to check whether the experience levels of the participants are reflected in their experiment performance.

4.3 Study Procedures

The quasi-experiment had six main steps that all participants should follow.

Step 1: Experiment presentation. At the beginning of the experiment session, the experiment instructor, the first author of this paper, showed a slide presentation explaining the Tic-Tac-Toe project and all the steps to run the experiment. The project was deliberately simple to run, consisting of pure HTML and Javascript code that could run directly from the filesystem in a browser, requiring no additional tooling or dependencies. Despite its simplicity, there are still real technical scenarios caveats in the assigned tasks, for instance the different treatment necessary in the Javascript data structures for unicode characters versus regular characters and the need to keep the scores even when a new game is started. Before starting any task, each participant was asked to read and fill out the consent document, providing their full name and a YES/NO authorization for the anonymized use of their data for academic research.

Step 2: Task 1 presentation. In the first task, the participants had to replace the traditional "X" and "O" symbols with cat and dog emojis. The participants were told that this task would be governed by the *Mastery-Aware* pipeline and that in order to do it they should follow its particular protocol. The students were also encouraged to try and break the rules, asking the agent to implement the task without adding a clear plan in the `TODO.md` artifact.

Step 3: Finishing the work. After finishing the first task, they should review the code changes and the `REPORT.md` artifact in order to fill the proof-of-mastery `REACTO.md` artifact. The developer could then ask the AI coding assistant if it was a good representation of the work done. The developer should review the work and update the artifact until the AI code assistant can give him a positive response. Finally, the developer would be able to commit the work and start the second task.

Step 4: Task 2 presentation. In the second task, the participants had to implement a scoreboard to track the players' wins. The experiment instructor explained how to disable the pipeline and presented the task, explaining that this particular task would no longer be restricted and that they could develop it the way they felt more comfortable.

Step 5: Socratic Debate. After accomplishing both tasks, the participants should engage in a Socratic debate with the AI code assistant, which was already configured to so through the leverage of custom instruction and agent skills [12, 13]. The debate would first ask predefined questions about the overall experience and finally socratic questions about their particular solution, for both Task 1 and Task 2. Once the Socratic debate was finished, the agent would finally generate the `SOCRATIC_REVIEW.md` artifact with the whole conversation and the verdict of the participants' success in defend their implementation.

Step 6: Submitting the work. At the end of the experiment, the participant could make comments to report any troublesome issue encountered along the experiment, and the agent verified that the consent document filled at the beginning of the session (Step 1) had been duly completed. Submissions affected by troublesome issues that would artificially contaminate the experimental results (e.g., participants who failed to properly store their work or interactions with the AI code agents) were later excluded during the data analysis, as detailed below.

4.4 Data Analysis Procedures

Once all participants had submitted their PULL REQUESTS, we proceeded by batch downloading and compiling them into a massive experiment report [10]. This report contains all the available data: `TODO.md`, `REPORT.md`, `REACTO.md`, and an equivalent version of `SOCRATIC_REVIEW.md` with the slight difference that this one would be the result of a debate about the overall experience, specific and dynamic questions for each task, and the perceived difference between Task 1 and Task 2 experience.

The dataset was really extent, contemplating the raw artifacts of 32 participants and 34 PULL REQUESTS, so we leveraged some of the best LLM models available [35, 36] for data analysis (*i.e.*, Claude Opus 4.7, Gemini Pro 3.1, and GPT-Codex 5.2) to translate, anonymize [31, 34], and analyze this data. A significant amount of submissions had to be rejected: 2 duplicated PULL REQUESTS from one participant were discarded from the dataset in favor of the their last submission, and 13 other submissions ended up being not measurable because participants failed to properly store their work or AI interactions, which left the key artifact `SOCRATIC_REVIEW.md` (the one that should contain the comparison results for Task 1 and Task 2) missing or incomplete. This left us with 19 accepted participants who submitted high-quality, measurable data. With this data in hands we could proceed with the data analysis.

Qualitative Analysis. We have conducted the qualitative evaluation of participant comprehension and artifact quality also with the support of AI agents. Their expertise in analyzing code structure and evaluating technical discussions make them good qualitative judges in automating error analysis and evaluating natural language reasoning [5]. This qualitative analysis brought us the results presented in Tables 3 and 4 in a detailed manner. We thoroughly inspected and double-checked all the scores presented in these tables.

Quantitative Analysis. Delegating the quantitative scoring of the participants to AI agents at first seemed like a considerable risk regarding accuracy and hallucinations. For that reason, this phase was conducted case by case for the 19 participants, where we could notice that the models were not relying on generative arithmetic, but were producing intermediate data analysis scripts to correctly compute statistical metrics (averages, deltas, standard deviations). To ensure the reliability of this data, all generated quantitative scores were manually double-checked by the authors against the participants' submitted artifacts, and the data was confirmed to be highly accurate and fair by their own judgment. This quantitative analysis brought us the results in Table 2.

5 Results and Discussion

This section presents and discusses the results of the quasi-experiment described in Section 4. We compare the pipeline-constrained condition (Task 1) with the unconstrained condition (Task 2) through three lenses: the composite Mastery Score, qualitative indicators from the Socratic debate, and observed design decisions.

5.1 Quantitative Mastery Scores

Table 2 reports the composite Mastery Score (0–100) across Module Explanation (0–45), Socratic Debate (0–35), and Design Justification (0–20). Individual scores are shown because variance itself is part of the result. The amount of scores given to these criteria was chosen based on their increasing relevance to cognitive engagement or debt. The ability to explain the modules involved in program changes (up to 45) is the most basic, essential aspects. The ability to defend the solution to peers – like in modern code review processes (up to 35) – or to herself is part of the required cognition process. The explication of design rationale (up to 20) is desirable but, in fact, neglected event in traditional (non-AI-assistance based) software development.

Table 2: Individual Mastery Scores per Task ($n = 19$).

PR	Exp.	Task 1 (Pipeline ON)				Task 2 (Pipeline OFF)				Δ
		Mod	Soc	Des	**T1**	Mod	Soc	Des	**T2**	
01	3y	45	35	20	**100**	34	26	15	**75**	−25
02	6y	23	35	10	**68**	0	26	5	**31**	−37
03	4y	45	35	20	**100**	45	35	20	**100**	0
04	4y	45	35	20	**100**	34	26	18	**78**	−22
06	3y	34	26	10	**70**	26	26	15	**67**	−3
07	3y	45	35	15	**95**	45	32	15	**92**	−3
12	3y	23	18	15	**56**	23	26	5	**54**	−2
13	2y	9	18	10	**37**	9	26	5	**40**	+3
14	4y	34	26	15	**75**	0	26	5	**31**	−44
17	3y	34	26	15	**75**	23	26	15	**64**	−11
18	3y	45	35	20	**100**	23	32	18	**73**	−27
20	4y	45	32	15	**92**	34	32	15	**81**	−11
21	4y	0	26	15	**41**	23	18	10	**51**	+10
25	10y	23	35	15	**73**	34	32	15	**81**	+8
28	9y	41	32	15	**88**	34	32	15	**81**	−7
30	6y	0	18	5	**23**	0	9	0	**9**	−14
31	10y	23	32	15	**70**	34	26	15	**75**	+5
33	10y	0	18	15	**33**	0	9	5	**14**	−19
34	10y	45	26	18	**89**	34	32	18	**84**	−5
Average		29.4	28.6	14.9	**72.9**	23.9	26.2	12.1	**62.2**	−10.7
Std. dev.		16.3	6.5	3.9	**24.1**	14.7	7.0	5.8	**25.5**	N/A

Mod = Module Explanation; *Soc* = Socratic Debate; *Des* = Design Justification.

The average Mastery Score fell from **72.9 points** under the pipeline to **62.2 points** without it, a decline of **10.7 points** (14.7% relative). The decrease was not driven by a small number of isolated cases: 15 of the 19 participants (78.9%) had a non-positive delta from Task 1 to Task 2. The standard deviation also increased from 24.1 to 25.5, suggesting that the unconstrained condition produced more polarized outcomes. Some participants maintained strong mastery without external enforcement, while others showed substantial losses once the specification and proof-of-mastery artifacts were removed. This pattern can be interpreted as a polarization effect: the unconstrained condition did not uniformly degrade performance, but instead amplified differences between participants, with some maintaining high mastery while others experienced substantial declines.

The extreme cases clarify this pattern. PR 14 had the steepest decline ($\Delta = -44$) and later summarized the Task 2 process as "I just let it do it". PR 2, despite six years of professional experience, fell by 37 points. PR 18, who achieved a perfect score in Task 1, dropped by 27 points in Task 2. In contrast, PR 3 obtained perfect scores in both tasks after identifying and correcting an AI-introduced bug before submission. These cases suggest that the decisive factor was not simply experience or tool capability, but whether the participant remained actively engaged in the reasoning process.

No clear relationship between programming experience and performance outcomes was observed. Participants with several years of experience were among those with the largest declines (e.g., PR 2 with 6 years, PR 14 with 4 years), while others maintained high performance. This suggests that experience alone did not predict resilience to the effects observed in the unconstrained condition. Similarly, the choice of AI model did not consistently explain participant outcomes. Comparable variability was observed within the same tool usage, indicating that differences in mastery were more closely associated with how participants engaged with the assistant than with the assistant itself. Taken together, the quantitative results indicate a consistent decline in performance across most participants, accompanied by increased variability, suggesting that the removal of structured workflow constraints is associated with both lower average mastery and less predictable outcomes.

5.2 Cognitive Debt Indicators

The Socratic debate complemented the numerical scores by exposing what participants could and could not explain about their own submissions. The strongest indicator was module explanation ability: whether participants could describe, without consulting the code, how the modified components interacted. Table 3 summarizes this distribution.

Table 3: Module explanation distribution ($n = 19$).

Level	Task 1 (Pipeline)	Task 2 (No Pipeline)	Δ
Yes (correct)	13 (68.4%)	6 (31.6%)	−7
Partial	3 (15.8%)	8 (42.1%)	+5
No	3 (15.8%)	5 (26.3%)	+2

The share of correct explanations fell from 68.4% to 31.6%, a decrease of **36.8 percentage points**. This is a different kind of measurement from the Mastery Score: instead of asking how much each participant scored on a continuous rubric, it asks how many participants crossed a specific comprehension threshold. The two measurements point in the same direction. Without the pipeline, fewer participants could give a complete account of the code they had just submitted. This result reinforces that functional correctness does not necessarily imply internalized understanding. Participants were often able to produce working solutions, yet were unable to explain how those solutions operated, indicating a disconnect between implementation and comprehension.

The failure modes were concrete rather than merely hesitant. PR 14 answered "No, I just let it do it". PR 2 reported not knowing exactly how the two JavaScript files interacted. PR 6 described

`script.js` as "backend" in a purely client-side application. PR 21 believed that the scoreboard reset after "New Game", although the implemented behavior accumulated scores across games. In each case, the submitted software could appear acceptable from the outside, while the participant's mental model diverged from the artifact. These cases illustrate a recurring pattern in which participants relied on AI-generated solutions without fully constructing a corresponding mental model, a phenomenon consistent with what can be described as cognitive bypass.

Autonomous debugging confidence showed a related shift. Table 4 reports the answers to the Socratic question: "Could you debug a critical error without AI?" The number of confident "Yes" answers remained stable, but the middle category eroded: four participants moved from "Partial" in Task 1 to an explicit "No" in Task 2. This shift in self-reported confidence aligns with the quantitative and qualitative findings, suggesting that reduced explanation ability is accompanied by reduced perceived ownership of the solution.

Table 4: Autonomous debugging confidence by task (n = 19).

Confidence	Task 1	Task 2	Δ
Yes	11 (57.9%)	11 (57.9%)	0
Partial	7 (36.8%)	3 (15.8%)	-4
No	1 (5.3%)	5 (26.3%)	+4

The wording of the answers is important. PR 12 stated that, without AI, "I might have to rewrite it from scratch". PR 13 answered "I think quite probably not". These statements indicate a loss of ownership that automated tests would not reveal. They also help explain why the Mastery Score decline is not merely a scoring artifact: participants themselves recognized that they would struggle to maintain the code independently.

The documentation artifacts showed the same dependency on structure. In Task 1, all 19 participants produced both `TODO.md` and `REACTO.md`, as required by the pipeline. In Task 2, after enforcement was removed, only 4 participants (21.1%) produced any reflective documentation voluntarily. This 79-percentage-point gap suggests that reflective practice did not yet behave as an internalized habit for most of the cohort. It appeared when the workflow demanded it, and largely disappeared when the demand was removed. This gap suggests that reflective practices did not behave as internalized habits for most participants, but instead depended on explicit workflow enforcement.

Finally, several submissions illustrated the difference between receiving a correct solution and learning from it. In five repositories (PRs 2, 3, 7, 12, and 17), the assistant correctly replaced `split("")` with `Array.from()` to handle multi-byte Unicode emoji characters. The fix was technically appropriate, but participants' explanations varied sharply. PR 12 admitted, "To be honest I have no idea [why]". PR 2 could describe the Unicode issue, but added that "I don't know how Copilot identified the problem". This pattern is consistent with the *Jagged Technological Frontier* [7]: the tool may solve a problem reliably while the developer's learning remains uneven. Across these observations, a consistent pattern emerges: correct solutions were frequently produced without a corresponding ability to explain or justify them, reinforcing that code correctness alone is not a reliable proxy for developer understanding.

Taken together, these findings address the research questions stated in Section 4. Regarding **RQ1**, cognitive debt could be systematically detected: independent measures (Mastery Score, explanation accuracy, debugging confidence, and documentation behavior) converged on the same phenomenon. Regarding **RQ2**, the evidence supports that structured desirable friction mitigates cognitive debt: the pipeline condition produced higher mastery, stronger explanations, and full reflective-documentation compliance, and all 19 participants reported that Task 1 gave them a deeper understanding of the system [10].

6 Implications

The results of this study suggest that structured interaction patterns with AI code assistants are associated with differences in developers' demonstrated understanding, consistency of outcomes, and engagement with reflective practices. Across both quantitative and qualitative measures, the presence of workflow constraints was linked to higher mastery scores, higher rates of correct explanations, and consistent use of reflection artifacts, while their absence was associated with lower average performance, increased variability, and reduced evidence of internalized understanding.

Structured Workflows and Cognitive Engagement. The findings indicate that requiring explicit specification before implementation and structured reflection afterward may support deeper cognitive engagement with AI-assisted development. Participants in the pipeline condition not only achieved higher average Mastery Scores, but were also more likely to correctly explain how system components interacted without consulting the code. These two measures, continuous performance scores and threshold-based explanation ability, converge on the same pattern: participants demonstrated stronger understanding when the workflow required them to articulate intent and reasoning.

The qualitative evidence reinforces this interpretation. In the unconstrained condition, several participants were unable to explain key aspects of their own implementations or expressed uncertainty about how their code functioned, even when the resulting behavior was correct. This suggests that the absence of structured reflection may allow development to proceed without corresponding mental-model construction. In contrast, the pipeline appears to encourage a form of active engagement in which developers must continuously reconcile generated code with their own understanding. While the results do not establish a causal mechanism, they are consistent with the interpretation that workflow-imposed "friction" can promote more deliberate reasoning, in line with *desirable difficulty* [3].

Variability, Individual Differences, and Experience. A central result is the increased variability observed in the unconstrained condition: although the average Mastery Score declined, the decrease was not uniform: some participants maintained high performance across both tasks, while others experienced substantial drops. Notably, this variability appears to be associated with how participants engaged with the AI assistant rather than with their experience level: several participants with multiple years of professional experience exhibited the largest declines, while others sustained strong performance by actively verifying and reasoning about generated solutions. This suggests that AI tools may amplify

existing work patterns rather than uniformly amplifying capability: developers who remain actively engaged can maintain understanding, while those who rely more passively on generated output may experience reduced comprehension.

Workflow Influence on Design and Development Practices. The findings suggest that workflow structure may influence not only cognitive outcomes but also technical decisions. In Task 1, participants who engaged in pre-implementation specification consistently adopted a display-layer approach that preserved the separation between game logic and presentation. In contrast, participants who did not articulate design intent beforehand were more likely to propagate changes throughout the internal state of the application.

This pattern indicates that requiring developers to specify goals and constraints before coding may make architectural considerations more explicit and actionable. By forcing a distinction between what should change and what should remain stable, the workflow may reduce the likelihood that design decisions are implicitly delegated to the AI assistant. Importantly, both strategies were functionally correct with respect to the task requirements, meaning that standard correctness checks would not distinguish between them. The observed difference therefore, reflects variation in design reasoning rather than task completion.

A similar dependency on workflow structure is observed in documentation practices. Reflective artifacts were consistently produced when required by the pipeline, but rarely appeared when the requirement was removed. This suggests that practices related to explanation and reflection were not yet internalized by most participants, and instead depended on explicit process enforcement. Together, these findings indicate that workflow design can shape both how developers think about code and how they structure it.

Implications for Practice, Tooling, and Education. For practitioners, the results suggest that incorporating a lightweight structure into AI-assisted workflows may help maintain understanding and code ownership. Practices such as requiring brief task specifications or post-implementation explanations could serve as checkpoints that ensure developers remain engaged with the reasoning behind generated solutions. The observed decline in explanation ability and debugging confidence in the absence of such practices indicates that, without intervention, correct code may not be accompanied by transferable understanding.

For tool builders, the findings highlight an opportunity to complement productivity-oriented features with support for comprehension-oriented workflows. Current AI-assisted development tools primarily optimize for speed and ease of code generation, but the results suggest that additional mechanisms (such as prompting developers to explain changes, justify design decisions, or validate assumptions) may help mitigate the loss of understanding observed in unconstrained settings. These features could be integrated into existing development environments as lightweight prompts or review stages.

For educators, the results reinforce the value of structured artifacts as both learning supports and assessment instruments. The combination of specification and reflection artifacts enabled the evaluation of understanding beyond functional correctness, revealing discrepancies between implemented behavior and participant explanations. This suggests that similar approaches could be used to assess learning in AI-assisted programming contexts, where traditional evaluation methods may fail to capture the extent of student comprehension.

Implications for Research. The study contributes an empirical approach to observing differences in developer understanding across workflow conditions, combining quantitative performance measures with qualitative indicators such as explanation ability and self-reported confidence. Future work should pursue replication in professional environments, longitudinal assessment of whether the observed differences persist (particularly during maintenance and debugging), and exploration of alternative workflow designs that balance AI-assisted productivity with sustained comprehension.

7 Threats to Validity

Internal and Construct Validity. The fixed task order (pipeline always before no-pipeline) was an intentional design choice to give Task 2 the advantage of recency; however, it limits causal isolation of the pipeline effect from familiarity with the codebase. A counterbalanced within-participants design would be required to control for this confound. A task-complexity equivalence analysis (cyclomatic complexity, Wilcoxon $p \approx 0.06$) is available in the extended report [10].

Participants may also have improved their interaction strategies with AI assistants during Task 1, introducing a learning or carryover effect that benefits Task 2 independently of the experimental condition. Conversely, fatigue or reduced motivation in the second task may have negatively influenced engagement and performance. Variability in adherence to the prescribed pipeline further introduces a threat to treatment fidelity, as participants may partially comply with or bypass required artifacts, leading to inconsistent exposure to the intervention. Moreover, participants' awareness of being observed may have influenced their behavior (Hawthorne effect), potentially increasing effort and engagement during the constrained condition.

Construct validity is further constrained by the operationalization of cognitive debt through the Mastery Score and Socratic review. While these instruments capture multiple dimensions of understanding, they rely on subjective evaluation and may not fully represent all aspects of developer cognition. The Socratic review process, while structured, may also introduce instrumentation bias, as variations in prompt phrasing or interpretation can affect evaluation consistency. Additionally, cognitive debt was assessed immediately after task completion, when the code was still fresh in memory. This timing may underestimate longer-term differences in retention and understanding; the observed effects may evolve significantly over time.

External Validity and Conclusion Validity. The experiment was conducted with a convenience sample of undergraduate and postgraduate computer science students at a single institution, which may limit generalizability to professional software engineers or different educational contexts. The use of a well-bounded Tic-Tac-Toe application further constrains ecological validity, as the pipeline's effectiveness in large-scale, multi-module, or production environments remains untested. More broadly, the simplicity and short duration of the tasks may not reflect the cognitive dynamics

of real-world, long-term software development. Finally, the diversity of AI models used by participants (eight variants) introduces variability in suggestion quality, which may influence performance independently of the pipeline.

8 Conclusion

This paper investigated the impact of AI-assisted development on developer cognition, with particular focus on the emergence of Cognitive Bypass and the resulting accumulation of cognitive debt. To address this challenge, we proposed the Mastery-Aware Pipeline, a structured workflow designed to reintroduce deliberate cognitive engagement through formal specification and post-implementation mastery verification.

To evaluate our proposed approach, an experimental study was conducted in which participants completed comparable programming tasks under two conditions: with and without the proposed pipeline. The methodology combined quantitative assessment using a multi-dimensional Mastery Score and qualitative analysis via Socratic review. The results consistently indicate that unconstrained AI delegation reduces developers' ability to explain, justify, and reason about their own code, while the introduction of structured "desirable friction" significantly improves these dimensions. In particular, participants using the pipeline demonstrated higher mastery scores, stronger conceptual understanding, and more deliberate architectural decisions. On the other hand, those operating without constraints exhibited signs of cognitive debt and reduced ownership of their solutions.

These findings suggest that productivity gains from AI-assisted development may come at the cost of diminished cognitive engagement if not properly managed, and that workflow design, not model capability, is the primary determinant of learning outcomes. In addition, by framing AI systems as collaborators that support, rather than replace, human reasoning, the proposed approach provides a foundation for more sustainable and cognitively aligned development practices.

Future work should extend this investigation through longitudinal studies to assess the persistence of cognitive debt over time, as well as large-scale industrial replications to evaluate the pipeline in real-world development environments. Additional research is also needed to automate mastery assessment and to explore adaptive mechanisms that calibrate the level of enforced friction based on developer expertise and task complexity. Together, these directions will help advance the design of AI-assisted workflows that balance efficiency with sustained developer understanding.

Artifact Availability

The artifacts related to this work are available at [9].

Acknowledgments

This work was partially supported by INES.IA (National Institute of Science and Technology for Software Engineering Based on and for Artificial Intelligence) www.ines.org.br, CNPq grant 408817/2024-0. The authors are also supported by FAPERJ (200.510/2023, 211.033/2019, 202.621/2019), CNPq (310391/2025-3), and CAPES/Proex (88887.373933/2019-00).

References

[1] Erfan Al-Hossami, Razvan Bunescu, Justin Smith, and Ryan Teehan. 2024. Can Language Models Employ the Socratic Method? Experiments with Code Debugging. In *Proceedings of the 55th ACM Technical Symposium on Computer Science Education V. 1.* Association for Computing Machinery, New York, NY, USA, 53–59. doi:10.1145/3626252.3630799

[2] Advait Bhat, Marianne Aubin Le Quéré, Mor Naaman, and Maurice Jakesch. 2026. Reactive Writers: How Co-Writing with AI Changes How We Engage with Ideas. In *Proceedings of the 2026 CHI Conference on Human Factors in Computing Systems (CHI '26).* Association for Computing Machinery, New York, NY, USA, Article 732, 21 pages. doi:10.1145/3772318.3791529

[3] Robert A. Bjork. 1994. Memory and Metamemory Considerations in the Training of Human Beings. In *Metacognition: Knowing About Knowing*, Janet Metcalfe and Arthur Shimamura (Eds.). MIT Press, 185–205.

[4] Birgitta Böckeler. 2025. *Understanding Spec-Driven Development: Kiro, spec-kit, and Tessl.* https://martinfowler.com/articles/exploring-gen-ai/sdd-3-tools.html Martin Fowler, Exploring Generative AI series. Accessed: 2026-06-30.

[5] Nadezhda Chirkova, Tunde Oluwaseyi Ajayi, Seth Aycock, Zain Muhammad Mujahid, Vladana Perlić, Ekaterina Borisova, and Markarit Vartampetian. 2026. LLM-as-a-qualitative-judge: automating error analysis in natural language generation. In *Proceedings of the First Workshop on Multilingual Multicultural Evaluation.* Association for Computational Linguistics, 99–132.

[6] Antonio Della Porta, Jonan Richards, Lucageneroso Cammarota, Stefano Lambiase, Fabio Palomba, and Mairieli Wessel. 2026. Ask, Then Think: Enhancing LLM Performance with Socratic Reasoning. *Available at SSRN 6167610* (2026).

[7] Fabrizio Dell'Acqua et al. 2023. *Navigating the Jagged Technological Frontier: Field Experimental Evidence of the Effects of AI on Knowledge Worker Productivity and Quality.* Technical Report 24-013. Harvard Business School.

[8] Yuyang Ding, Hanglei Hu, Jie Zhou, Qin Chen, Bo Jiang, and Liang He. 2024. Boosting Large Language Models with Socratic Method for Conversational Mathematics Teaching. In *Proceedings of the 33rd ACM International Conference on Information and Knowledge Management* (Boise, ID, USA) *(CIKM '24).* Association for Computing Machinery, New York, NY, USA, 3730–3735. doi:10.1145/3627673.3679881

[9] Neto et al. 2026. Brains-Back TicTacToe Replication Package. https://github.com/viccsneto/brains-back-tictactoe-replication

[10] Neto et al. 2026. Tic-Tac-Toe experiment report. https://github.com/viccsneto/brains-back-tictactoe-replication/blob/master/tictactoe-replication-package/tic-tac-toe-experiment-report/experiment_report.pdf

[11] Lucile Favero, Juan Antonio Pérez-Ortiz, Tanja Käser, and Nuria Oliver. 2024. Enhancing Critical Thinking in Education by Means of a Socratic Chatbot. In *International workshop on AI in education and educational research.* Springer, 17–32.

[12] GitHub. 2026. About Agent Skills for GitHub Copilot. GitHub Documentation. https://docs.github.com/en/copilot/concepts/agents/about-agent-skills

[13] GitHub. 2026. About Custom Agents for GitHub Copilot. GitHub Documentation. https://docs.github.com/en/copilot/concepts/agents/coding-agent/about-custom-agents

[14] Martin Gjoreski, Bhargavi Mahesh, Tine Kolenik, Jens Uwe-Garbas, Dominik Seuss, Hristijan Gjoreski, Mitja Luštrek, Matjaž Gams, and Veljko Pejović. 2021. Cognitive Load Monitoring With Wearables–Lessons Learned From a Machine Learning Challenge. *IEEE Access* 9 (2021), 103325–103336. doi:10.1109/ACCESS.2021.3093216

[15] Lucian Gonçales, Kleinner Farias, Bruno da Silva, and Jonathan Fessler. 2019. Measuring the Cognitive Load of Software Developers: A Systematic Mapping Study. In *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC).* 42–52. doi:10.1109/ICPC.2019.00018

[16] Junda He, Christoph Treude, and David Lo. 2025. LLM-Based Multi-Agent Systems for Software Engineering: Literature Review, Vision, and the Road Ahead. *ACM Transactions on Software Engineering and Methodology* 34, 5, Article 124 (2025). doi:10.1145/3712003

[17] Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang. 2024. Large language models for software engineering: A systematic literature review. *ACM Transactions on Software Engineering and Methodology* 33, 8 (2024), 1–79.

[18] Nataliya Kosmyna. 2025. Your Brain on ChatGPT: Project Overview. MIT Media Lab. https://www.media.mit.edu/projects/your-brain-on-chatgpt/overview/

[19] Nataliya Kosmyna et al. 2025. Your Brain on ChatGPT: Accumulation of Cognitive Debt when Using an AI Assistant for Essay Writing Task. *arXiv preprint arXiv:2506.08872* (2025). https://arxiv.org/pdf/2506.08872

[20] Zhixian Christopher Liding, Michael Osmolovskiy, Harshith Lanka, Ronnie Howard, Nimisha Roy, and Rodrigo Borela. 2026. To Tell or to Ask? Comparing the Effects of Targeted vs. Socratic AI Hints. In *Proceedings of the 57th ACM Technical Symposium on Computer Science Education V. 2.* Association for Computing Machinery, New York, NY, USA, 1423–1424.

[21] Lujin Mao, Linyuan Dong, Wenan Li, Xiangen Hu, Kun-Pyo Lee, and Zhibin Zhou. 2026. Designing Scaffolding Cards to Facilitate LLM-Based Socratic Instruction:

An Exploratory Study of Response Strategies to Support Learning. In *Proceedings of the 2026 CHI Conference on Human Factors in Computing Systems*. Association for Computing Machinery, New York, NY, USA, 1–23. doi:10.1145/3772318.3791696

[22] Shakked Noy and Whitney Zhang. 2023. Experimental Evidence on the Productivity Effects of Generative Artificial Intelligence. *Science* 381, 6654 (2023), 187–192. doi:10.1126/science.adh2586

[23] James C Overholser and Eleanor Beale. 2023. The Art and Science Behind Socratic Questioning and Guided Discovery: A Research Review. *Psychotherapy Research* 33, 7 (2023), 946–956.

[24] Neil Perry, Megha Srivastava, Deepak Kumar, and Dan Boneh. 2023. Do Users Write More Insecure Code with AI Assistants?. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security (CCS '23)*. Association for Computing Machinery, New York, NY, USA, 2785–2799. doi:10.1145/3576915.3623157

[25] Zeeshan Rasheed, Malik Abdul Sami, Muhammad Waseem, Kai-Kristian Kemell, Xiaofeng Wang, Anh Nguyen, Kari Systä, and Pekka Abrahamsson. 2024. AI-Powered Code Review with LLMs: Early Results. *arXiv preprint arXiv:2404.18496* (2024).

[26] Giuseppe Romeo and Daniela Conti. 2026. Exploring automation bias in human–AI collaboration: a review and implications for explainable AI. *Ai & Society* 41, 1 (2026), 259–278.

[27] Giovanni Rosa, David Moreno-Lumbreras, Gregorio Robles, and Jesús M González-Barahona. 2026. Understanding Specification-Driven Code Generation with LLMs: An Empirical Study Design. *arXiv preprint arXiv:2601.03878* (2026).

[28] Sarah (@sarahscode). 2017. REACTO: Technical Interview Prep the Fullstack Way. Medium. https://medium.com/@sarahscode/reacto-technical-interview-prep-the-fullstack-way-706929a44e90

[29] William R. Shadish, Thomas D. Cook, and Donald T. Campbell. 2002. *Experimental and Quasi-Experimental Designs for Generalized Causal Inference*. Houghton Mifflin, Boston, MA.

[30] Margaret-Anne D. Storey. 2026. From Technical Debt to Cognitive and Intent Debt: Rethinking Software Health in the Age of AI. *CoRR* abs/2603.22106 (2026). arXiv:2603.22106 doi:10.48550/ARXIV.2603.22106

[31] Bolun Sun, Yifan Zhou, and Haiyun Jiang. 2024. Empowering Users in Digital Privacy Management through Interactive LLM-Based Agents. *arXiv* (2024). doi:10.48550/arxiv.2410.11906

[32] Mira Suryani, Harry Budi Santoso, Martin Schrepp, Rizal Fathoni Aji, Setiawan Hadi, Dana Indra Sensuse, Ryan Randy Suryono, and Kautsarina. 2024. Role, Methodology, and Measurement of Cognitive Load in Computer Science and Information Systems Research. *IEEE Access* 12 (2024), 190007–190024. doi:10.1109/ACCESS.2024.3514355

[33] Wannita Takerngsaksiri, Jirat Pasuksmit, Patanamon Thongtanunam, Chakkrit Tantithamthavorn, Anastasiia Tkalich, Michael Donoghue, Marcus Oertle, and Brendan Maclean. 2025. Human-In-The-Loop Software Development Agents. In *Proceedings of the 47th IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP '25)*. IEEE, 342–352.

[34] VenturusBR. 2025. Guardians of the data: NER and LLMs for effective medical record anonymization in Brazilian Portuguese. *Frontiers in Public Health* 13 (2025). doi:10.3389/fpubh.2025.1717303

[35] Colin White, Samuel Dooley, Manley Roberts, Arka Pal, Benjamin Feuer, Siddhartha Jain, Ravid Shwartz-Ziv, Neel Jain, Khalid Saifullah, Sreemanti Dey, Shubh-Agrawal, Sandeep Singh Sandha, Siddartha Venkat Naidu, Chinmay Hegde, Yann LeCun, Tom Goldstein, Willie Neiswanger, and Micah Goldblum. 2025. LiveBench: A Challenging, Contamination-Free LLM Benchmark. In *The Thirteenth International Conference on Learning Representations*.

[36] Colin White, Samuel Dooley, Manley Roberts, Arka Pal, Ben Feuer, Siddhartha Jain, Ravid Shwartz-Ziv, Neel Jain, Khalid Saifullah, Siddartha Naidu, et al. 2024. Livebench: A challenging, contamination-free llm benchmark. *arXiv preprint arXiv:2406.19314* 4 (2024), 2.

[37] Robert E. Wray, James R. Kirk, and John E. Laird. 2025. Applying Cognitive Design Patterns to General LLM Agents. In *Proceedings of the 18th International Conference on Artificial General Intelligence (AGI 2025)*. Springer. doi:10.1007/978-3-032-00800-8_28