Grasping AI Reliance in Program Comprehension and Coding through the AIRELI Persona Taxonomy

Tarek Alakmeh
tarek.alakmeh@uzh.ch
University of Zurich
Zurich, Switzerland

Norman Anderson
normananderson@uvic.ca
University of Victoria
Victoria, Canada

Victoria Jackson
v.jackson@soton.ac.uk
University of Southampton
Southampton, UK

Guilherme Vaz Pereira
guilherme.v003@edu.pucrs.br
Pontifícia Universidade Católica do Rio Grande do Sul
Porto Alegre, Brazil

Umit Akirmak
uakirmak@uvic.ca
University of Victoria
Victoria, Canada

Anthony Estey
Aestey@uvic.ca
University of Victoria
Victoria, Canada

Rafael Prikladnicki
rafael.prikladnicki@pucrs.br
Pontifícia Universidade Católica do Rio Grande do Sul
Porto Alegre, Brazil

André van der Hoek
andre@ics.uci.edu
University of California
Irvine, USA

Margaret-Anne Storey
mstorey@uvic.ca
University of Victoria
Victoria, Canada

Thomas Fritz
fritz@ifi.uzh.ch
University of Zurich
Zurich, Switzerland

Abstract

Artificial Intelligence (AI) assistance has become an integral part of software development, helping developers plan, explain, and generate code. As the boundary between human agency and AI reliance blurs, traditional measures of program comprehension, such as task success or completion time, increasingly capture AI effectiveness rather than the depth of human understanding. Without a better understanding of how developers rely on AI and how it replaces human expertise, it is difficult to assess its short- and long-term effects on comprehension and capability.

We conducted a controlled study with 21 participants working on two realistic change tasks, with or without AI assistance. Using quantitative and qualitative data from screen recordings, performance metrics, and questionnaires, we performed a thematic analysis and derived nine key characteristics that informed three AI reliance personas: self-sufficient, understanding-gated, and AI-steered developers. Analyzing participants through this persona lens revealed substantial differences in comprehension and capability that aggregate comparisons between AI and No AI conditions masked. Self-sufficient developers demonstrated deep understanding, understanding-gated developers retained conceptual understanding but relied on AI for execution, and AI-steered developers completed tasks quickly yet without meaningful comprehension. These findings highlight the importance of accounting for AI reliance, as short-term AI-assisted productivity gains can mask a growing *comprehension debt*, where cognitive work is outsourced to AI at the expense of human expertise and sustainable skill development.

CCS Concepts

• **Software and its engineering → Software creation and management**.

Keywords

Program comprehension, code comprehension, AI reliance, controlled study

ACM Reference Format:
Tarek Alakmeh, Norman Anderson, Victoria Jackson, Guilherme Vaz Pereira, Umit Akirmak, Anthony Estey, Rafael Prikladnicki, André van der Hoek, Margaret-Anne Storey, and Thomas Fritz. 2026. Grasping AI Reliance in Program Comprehension and Coding through the AIRELI Persona Taxonomy. In *34th IEEE/ACM International Conference on Program Comprehension (ICPC '26), April 12–13, 2026, Rio de Janeiro, Brazil.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3794763.3794804

This work is licensed under a Creative Commons Attribution 4.0 International License.
ICPC '26, Rio de Janeiro, Brazil
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2482-4/2026/04
https://doi.org/10.1145/3794763.3794804

1 Introduction

Generative AI has become a routine companion in software development: developers ask assistants to outline plans, explain unfamiliar code, and generate patches that compile on the first attempt [3, 13, 29]. As this assistance becomes ubiquitous, a core question emerges: what exactly does a human developer (still) comprehend? Most traditional program comprehension studies use proxies for

understanding, such as task success or time-on-task [48]. Yet, when developers outsource the planning and execution of a task to AI assistants, these proxies reflect more of the effectiveness of the AI assistant than the depth of human comprehension.

Recent studies have examined how AI assistance influences program comprehension by comparing developers' performance on small controlled tasks with AI to a baseline without AI [11, 44, 50]. These comparisons provide valuable insights but are limited to short code snippets and simplified contexts, assessing AI assistance only at an aggregate level (AI vs no AI). In practice, developers integrate AI assistance into their workflows in diverse ways: as a tutor for refining plans, as a source of "glue" code, or even as the primary driver of a solution. This diversity highlights the need to characterize not just how AI is used, but also how much developers rely on it. Such reliance has become so embedded that AI assistance can no longer be meaningfully isolated, much like one would not remove an IDE to study its impact on programming productivity today. Instead, understanding program comprehension and developer behavior in the age of AI requires examining patterns of reliance and the substitution of human expertise with AI, as these dynamics shape both what developers understand and how that understanding evolves. This poses the following research questions:

RQ1 How does AI reliance manifest during program comprehension and coding?

RQ2 What are the implications of AI reliance for program comprehension?

To investigate these questions, we conducted a controlled study with 21 participants working on two realistic web application development tasks, with and without access to generative AI assistance. The study tasks involved modifying a larger codebase and coordinating changes across frontend and backend components, activities that, before the advent of generative AI, typically required a conceptual understanding of how system components interact. Throughout the study, we collected a rich set of qualitative and quantitative data, including performance metrics, screen recordings, and responses to questionnaires assessing comprehension and experience. Following a thematic analysis approach, we derived nine characteristics that capture the most prominent patterns in the data and informed three AI reliance personas: self-sufficient, understanding-gated, and AI-steered developers. Viewing developer behavior through this persona lens offered a more nuanced understanding of AI reliance and the outsourcing of comprehension and capability to the AI. While *self-sufficient* developers understood planned, and executed their changes, *understanding-gated* developers formed a conceptual understanding and authored their plan, but leaned on AI for execution, and *AI-steered* developers thrived with AI yet struggled without it and lacked comprehension.

These findings highlight the importance of an AI reliance-aware perspective in future studies of program comprehension and developer performance, as aggregate comparisons between AI and No AI conditions may mask critical differences in human understanding and capability. In particular, while extensive AI reliance may lead to short-term productivity gains, it can create a growing *comprehension debt*, a gap between the code that developers can produce with AI and what they genuinely understand. Over time, this debt may erode core competencies and long-term skill development. This paper makes the following key contributions:

- **AI Reliance Persona Taxonomy.** We introduce a taxonomy comprising three AI reliance personas and nine associated characteristics, derived from an in-depth analysis of data collected in a controlled study. The taxonomy offers a reusable framework for nuanced analysis of developer behavior, code comprehension, and capability during software change tasks.
- **Empirical insights.** Using data from 21 participants performing realistic web application change tasks, we provide observations on how varying levels of AI reliance substantially affect comprehension and capability, providing empirical evidence for the value of the taxonomy.
- **Implications for research and practice.** We discuss how AI reliance shapes program comprehension and developer performance, and how early AI-enabled productivity gains may hide comprehension debt, posing long-term risks for junior developers and their learning trajectories.

We provide the dataset, the taxonomy with additional details and participant mappings, and the replication package [2] as a web app companion at https://aireli.hasel.dev.

2 Related Work

There is extensive research on program comprehension and, more broadly, on how developers understand and modify code, covering a wide range of aspects such as the cognitive processes involved [7, 38, 45], the mental models formed during comprehension [18, 24], and the influence of code readability and legibility on understanding [33]. Many of these studies, particularly those on program comprehension, tend to use small, researcher-created, self-contained code snippets that participants can grasp quickly, thereby avoiding tasks requiring architectural understanding across files or modules, as demonstrated by the systematic mapping study of Wyrich et al. [48]. With modern AI assistants now capable of solving such isolated snippet tasks with ease [3, 21, 37, 53], there is an increasing need to design comprehension experiments that involve larger, more complex, and realistic codebases. While recent educational studies have begun to explore related challenges and opportunities [41], there is a continued need for empirical work on comprehension in larger, interconnected systems, particularly as AI assistants become increasingly ubiquitous and proficient at handling small-scale code tasks.

Prior work across software engineering and computing education already showed that developers use AI not only to generate code but also for conceptual understanding, planning, and debugging [15, 19, 22, 23]. A recurring characterization is a bimodal pattern of use. Developers turn to AI for *acceleration* when they already know the next step and want efficiency on routine actions, and for *exploration* when they face uncertainty and seek candidate directions or examples [3]. Framing usage in this way might explain why reported outcomes on speed and productivity vary widely [4, 14, 34, 37]. Alongside potential productivity benefits, multiple types of risk are repeatedly raised by studies investigating the effects of AI. Security analyses show that AI generated code often

contains known vulnerabilities, which raises the bar for validation and comprehension rather than adoption based on surface plausibility [20, 31, 36]. Concerns about functional correctness are similarly widespread [9, 26, 28], as are worries about maintainability and long-term quality, including style consistency, dependency management, and readability for future change [12, 16, 27]. Across these threats, researchers emphasize calibration of trust as a mediating factor that shapes whether outputs are scrutinized, tested, and adapted or accepted with minimal evaluation [8, 41, 46].

Studies of experienced engineers describe triage practices that rely on architectural knowledge, idioms, and tests to probe assumptions and fit AI suggestions into existing constraints [4, 30, 43]. In educational and early career contexts, by contrast, empirical work reports difficulties in formulating and refining prompts, in recognizing model errors, and in recovering from misleading outputs. These challenges can foster automation bias and illusions of competence, where fluent output is mistaken for understanding [32, 40, 52]. For less experienced developers, AI reliance can therefore outpace human comprehension unless evaluation strategies are made explicit and supported. Early work showed that designing such assistants in the IDE improved success and satisfaction on code understanding tasks, suggesting that reliance risks can be mitigated when the assistant is situated within comprehension activities during code generation [30, 51]. At the same time, using LLMs to enhance understanding should be used with caution: evaluations of ChatGPT on program comprehension questions reveal weaknesses analogous to novices, including difficulty following execution traces and missing subtle but important details [25]. Evidence with novices further shows that initial trust can be high and plausibility can be mistaken for correctness, strengthening the case for interfaces that surface uncertainty and rationale during comprehension [39].

Taken together, the literature situates reliance as shaped by both task demands and developer expertise. Especially for novices, relying on AI can obscure gaps in understanding and amplify risks around security, correctness, and maintainability unless evaluation and verification are emphasized. Yet, despite growing interest in how AI assistance influences developer behavior, we still lack a nuanced understanding of how reliance on AI affects human comprehension during realistic, multi-file development tasks, a gap that this work begins to address.

3 Methodology

The goal of this study was to examine how reliance on generative AI affects task performance, with a particular focus on comprehension, overall execution and learning effects across similar tasks. To address this goal, we conducted a controlled lab experiment with 21 participants, each working on a sequence of two tasks (T_1, T_2), while altering whether or not the participant was allowed to use generative AI for the first task T_1. Specifically, we randomly assigned each participant to one of two groups: one group was allowed to perform T_1 with generative AI (i.e., using the free tier of ChatGPT with the GPT-4o model), while the other group had to perform T_1 without it and was only given access to traditional web use such as search, online documentation, or Stack Overflow. For T_2, both groups were not allowed to use any external help (i.e., no generative AI and no web use). We designed the two tasks to be similar yet distinct: T_2 mirrored T_1 in structure and complexity, allowing us to observe learning effects across tasks, while differing enough to prevent simple replication and require meaningful adaptation. Overall, this design allowed us to compare generative AI use *between* subjects for T_1 and also to assess how much learning/comprehension effects carried over from task T_1 to T_2 *within* subjects. Since T_2 was designed to capture carry-over effects from T_1, a fully counterbalanced or cross-over design (e.g., allowing ChatGPT in T_2) would blur carry-over effects with the effects of using ChatGPT during T_2. The study was approved by our institutional ethics board and participants were reimbursed US$50 for their participation irrespective of task success or time.

3.1 Participants

We recruited undergraduate and graduate students in computer science or related fields via email, word of mouth, and advertising in a software engineering project course, where students develop web applications. To be eligible, participants had to have at least basic experience in web development using TypeScript. Before participating, they received detailed procedural information, completed a demographics and programming experience survey, and signed an IRB-approved consent form. A total of $N = 22$ participants (18 male, 4 female, aged 24.0 ± 2.0 years) took part in the study. They had an average of 1.4 ± 1.9 years of professional and 4.6 ± 3.7 years of total programming experience; 10 were Bachelor's and 12 Master's students. One participant was excluded for noncompliance with our written and verbal instructions, leaving 21 for analysis.

3.2 Procedure

The study lasted a maximum of two and a half hours and was conducted in person. It began with an onboarding session in which participants set up the provided GitHub Codespace and verified that they could start the server and preview the web application. We confirmed that each environment was functioning properly before proceeding.

In the main part of the study (see Figure 1), participants were asked to work on two tasks (first T_1 followed by T_2). For each task, they received a description with acceptance criteria (for validating successful completion), an overview of the repository structure, and local run instructions in a README. After each task, participants were asked to complete two questionnaires: one on task experience and one assessing comprehension. Participants who did not complete a task within two hours were asked to proceed to the questionnaires to keep the total procedure time below 2.5 hours.

Participants were randomly assigned to the AI or No AI group in a rolling fashion. Because recruitment was continuous (with participants added over time and occasional no-shows), we generated randomized assignments in batches throughout the study using a computer script. For the second task T_2, both groups had to complete it without generative AI. Copilot and other code completion tools were disabled throughout the study. We verified adherence to the assigned conditions and use of tools via screen recordings. Statistical comparisons of self-reported familiarity scores provided prior to participation confirmed that both groups were balanced across study programs ($p = 0.239$), frontend familiarity ($p = 0.280$), and backend familiarity ($p = 0.115$), with no significant differences.

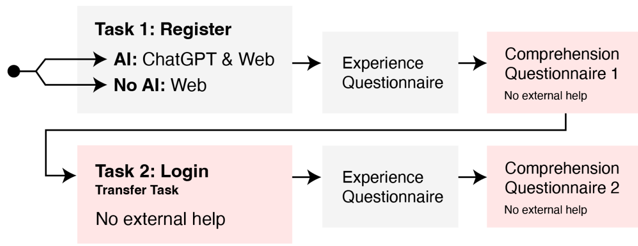

Figure 1: Overview of study procedure (red boxes indicate no use of external help such as using the web or generative AI).

3.3 Tasks

Rather than using short, isolated programming exercises, we designed realistic web application development tasks that required participants to navigate and modify a larger, multi-file codebase, coordinating changes across both frontend and backend components. The frontend consisted of 234 files based on an enterprise-grade ecommerce starter template [5]. The backend comprised 13 files intended to mimic the initial files of a new backend system. We deliberately designed the tasks to mirror situations that, prior to the advent of generative AI, traditionally demanded a deeper understanding of how different components interact with each other.

The study emulated a realistic junior developer assignment, where a newcomer joins an existing web application and completes a first ticket spanning frontend and backend. The task involved replacing calls to an outsourced user management service with a self-hosted backend. The codebase was a TypeScript/JavaScript monorepo with a ReactJS/NextJS frontend, an ExpressJS backend, and a SQLite database. To simulate typical onboarding, participants received a detailed README inside the VS Code IDE, serving as both documentation and task description, providing key entry points into the codebase (e.g., the routing file, relevant controller, user model, and frontend actions) while still requiring them to understand how these parts interact. Framing the work as a migration allowed us to provide substantial code and documentation, keeping the study feasible within a short session while still exposing participants to realistic vertical changes requiring architectural understanding and coordinated edits across layers.

Task T_1: Register Functionality. The first task required implementing user registration across the full stack, enabling users to register and receive a confirmation message. In the backend, this required a corresponding extension of the user model and controller logic, as well as a new API route. In the frontend, the user actions had to be updated to use the new internal client. The task intentionally spanned multiple modules, requiring navigation of entry points and data flow while preserving design consistency. In total, a solution required modifying at least five files (sample solution can be found in [2]) and affected various components, as illustrated in the system architecture sketch of task T_1 by one of our participants in Figure 3.

Task T_2: Login Functionality. The second task, also full-stack, involved implementing user login. It required adding backend routing, controller logic, user model extensions, and updating frontend actions to call the new endpoint through the internal client. This task mirrored T_1 in structure, intentionally designed to enable transfer of understanding from T_1 while differing enough to prevent simple replication or copy-and-paste solutions. Similar to T_1, this task required modifying at least five files in various components.

3.4 Piloting

We piloted the study materials with four undergraduate and graduate students with varying levels of web development experience (two reported to have barely any experience, one reported some experience, and one reported high proficiency) to calibrate task complexity and timing within the 2.5-hour limit. Based on feedback, we retained the cross-file and cross-layer structure of the tasks but improved feasibility through targeted adjustments. Specifically, we sketched function signatures for key entry points, added clarifying comments to indicate nonessential code for Task T_1, and expanded the README with clearer directions to relevant files. We avoided further simplification, such as providing near-complete templates, to preserve the need for architectural/structural comprehension.

3.5 Data Collection

Throughout the study, we collected quantitative and qualitative data. To analyze developer behavior, we recorded participants' screens during the study sessions, their code changes for each task and the time it took to complete each task, and, for the AI group, saved time-stamped ChatGPT transcripts from the provided account.

We further collected participants' responses to a pre-study demographics and programming experience survey, as well as to the experience and comprehension questionnaires administered after each task. The *experience questionnaire* captured participants' rating of perceived task difficulty on a five-point Likert scale, cognitive effort using the single-item Paas Mental Effort Scale, and confidence to continue working in the same codebase on a five-point Likert scale, complemented by an open-ended item on any major difficulties participants encountered. The *comprehension questionnaire* assessed architectural and implementation understanding through (1) a sketching task and (2) K-Prim multiple-choice questions focusing on task-relevant components and relationships. First, participants were asked to sketch the system architecture, capturing task-relevant components and the relationships between them. Second, participants were asked to answer K-Prim style multiple-choice questions targeting concrete implementation details. Each multiple-choice item presented four statements about the solution, along with an "I don't know" option to discourage guessing. Alongside correct statements, we included seemingly plausible but incorrect statements to test for actual system comprehension rather than general-purpose-case guessing. We graded sketches on a 4-point rubric, 2 points for components and 2 for relationships, with partial points decreasing as required components or relationships were missing or incorrect. The same instruments were administered after both tasks to enable within-subject comparisons of perceived effort and comprehension between conditions. Full details can be found at our companion web app (or replication package) online [2].

3.6 Data Analysis

We followed a thematic analysis approach for the collected data, deriving characteristics (subsection 4.1) and, in a second step, higher-level AI reliance (AIRELI) personas from it (subsection 4.2). We then used the personas as a lens to look at developer behavior and performance (subsection 5.1), before using the persona characteristics as a lens and mapping them to the key stages of the code change process (subsection 5.2).

4 Characteristics and Personas

Determining how much a developer relies on AI during change tasks is challenging, as reliance cannot easily be measured through indicators such as AI usage metrics or self-reports. Most of our participants reported frequent use of AI for code generation, explanation, and error finding. However, high AI usage can coexist with low reliance when the developer remains in charge of understanding and validation, while conversely, low reported usage can hide deep dependence in critical moments. This raises the question of how reliance manifests itself.

In a first step, we therefore created a consolidated *case file* for each participant that contained: (i) demographics information, including self-reported AI use and familiarity the task-relevant technologies, (ii) task outcome indicators for T_1 and T_2, (iii) comprehension evaluations (including multiple choice and system sketch responses), (iv) screen recordings summarized as written observations with a focus on AI use (e.g., point in time of first AI contact, conversation durations, frequency), (v) code change/diffs, (vi) AI chat logs for participants in the AI condition, (vii) self-reported assessments of task difficulty from the experience questionnaires.

Thematic Analysis. To explore AI reliance, we then followed a reflexive thematic analysis (RTA) [6] approach focusing on how participants formed understanding and interacted with AI. Specifically, we triangulated the different data sources in each case file in our analysis. Two authors conducted iterative, interpretive coding with a shared analytical focus on three stages of the change process: (1) system understanding, (2) plan formulation, and (3) plan execution. We began with repeated familiarization of the case files, then performed open coding on an initial subset of participants to generate candidate codes, such as 'frustration' or 'architecture overload'. Through regular analytic meetings and memoing, we refined code definitions and clarified boundaries with brief inclusion and exclusion notes. We then applied the refined codes across all cases, iteratively revisiting earlier cases when code definitions evolved. We stopped refining when no substantively new codes emerged and code boundaries stabilized. The case-file construction allowed us to turn raw artifacts into analyzable signals and ensured that the evidential basis of each code could be traced back to specific observations, edits, or questionnaire items.

In two steps, we first derived a set of characteristics based on the codes (Section 4.1), and then secondly, three personas (Section 4.2) using these characteristics. An overview of the process is illustrated in Figure 2. To support replicability, our web companion app and replication package include the participant case-files, the characteristic definitions, and the participant-to-characteristic mapping, enabling other researchers to apply the same analytic pipeline to new datasets [2].

4.1 Characteristics

We organized recurring codes into characteristics and validated each characteristic against multiple instances across participants. We required "clear evidence" (e.g., repeated behavioral indicators in the screen recording, corroborating questionnaire statements, or consistent patterns in code diffs), and we only assigned a characteristic when the case file provided such evidence.

Overall, we derived nine characteristics (see Table 1) that capture the most prominent patterns we observed in the raw data. These characteristics represent a non-exhaustive and non-exclusive set and can roughly be grouped into three categories based on their origin and focus: system orientation, AI usage style, and demographics. For example, a *'Strong Cartographer'* (**SC**) denotes a participant who drew a high-quality system sketch (e.g., Figure 3, *Comprehension Performance*), scoring higher than 3.5 on our grading scheme. An *'Instant AI Developer'* (**IAI**), on the other hand, refers to a participant that immediately used AI to delegate the implementation without first building an understanding, as observed in the screen-recordings (*Observations*) and the *AI Chat Logs* among other codes.

Multiple characteristics could apply to a single participant. For example, participant P5 used AI instantly (**IAI**), continued to use AI with high frequency (**HAI**), reported that it was difficult to connect frontend and backend (**AO**) and that issue persisted throughout the task leading to disorientation (**DD**) with many missing links in their system sketch. We only assigned characteristics when there was clear evidence to support it. In some cases, the evidence was ambiguous, partly due to variation in participants' free-form responses about experienced difficulties and the availability of AI. For instance, while some participants explicitly commented on the presence or absence of AI (based on the AI or No AI conditions) and its impact on their implementation and comprehension, others did not.

4.2 AI Reliance (AIRELI) Personas

After deriving characteristics, we conducted affinity mapping of co-occurring characteristics and identified three higher-level clusters of participants in relation to AI reliance: self-sufficient, understanding-gated, and AI-steered developers (see Table 2). Disagreements during the affinity mapping process were resolved through discussion and by returning to the underlying case-file evidence. The three identified clusters represent *personas* that can be differentiated by how participants approached key stages of the code change process—(1) system understanding, (2) change plan formulation, and (3) change plan execution—and by the extent to which they relied on AI during these stages. *Self-Sufficient* developers are able to understand the system (1), formulate a plan (2), and execute it on their own (3); AI, when used, is complementary (e.g., speed or boilerplate), not a crutch. *Understanding-Gated* developers are able to understand (1) and plan (2) without AI, but may rely on AI to execute (3); some also use AI as a tutor to refine planning, yet the developer remains in charge of the plan and its verification. *AI-Steered* developers are not consistently able to understand (1) and/or plan (2) without AI and therefore rely on AI to drive execution (3). In these cases, the AI effectively takes charge of solving the

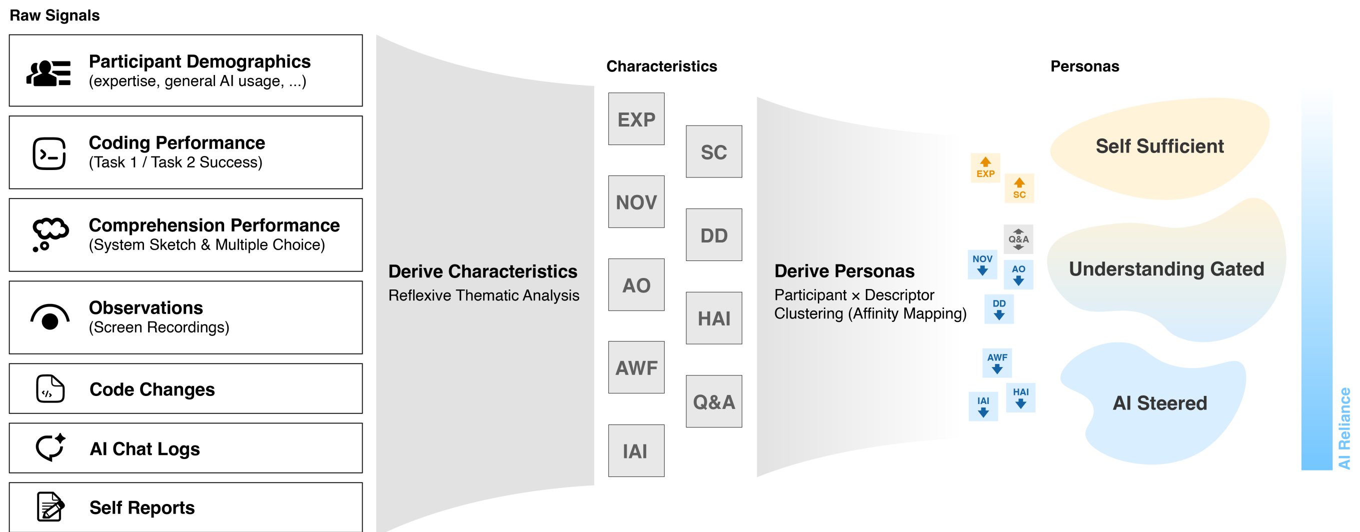

Figure 2: Thematic analysis steps: from raw input signals over participant Characteristics to Personas. Refer to Table 1 for definitions and descriptions of Characteristic abbreviations.

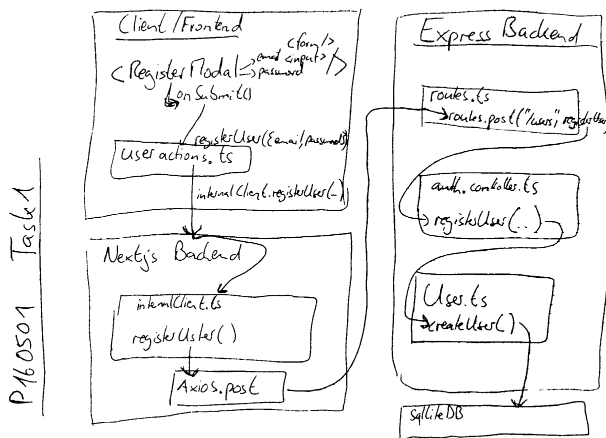

Figure 3: A participant's system sketch showing task-relevant components and their relationships for task T_1.

task, and the developer follows the AI's instructions by inserting the generated code at the right places in the code.

To operationalize AI reliance, we formulated three guiding questions: (1) Can the developer form a reasonable understanding of the task-relevant parts of the system? (2) Can they formulate an implementation plan without outsourcing the planning to AI? (3) Can they execute that plan, including implementation and debugging? A developer's level of reliance is reflected in the extent to which this three-step process fails or depends on AI support.

These guiding questions also provide the analytical lens for relating the identified characteristics to the personas. Patterns such as *Strong Cartographer* and *Expert* typically indicate solid understanding, planning, and autonomous execution, often corresponding to the self-sufficient developer. In contrast, combinations such as *Disoriented Developers*, *Architecture Overloaded*, *AI Withdrawal Frustration*, or *Instant AI Developers* point to gaps in understanding or planning and a tendency toward AI-led execution (3), aligning with the AI-steered developer. *Q&A Style Co Pilots* often fall in between: planning is developer-led, while execution is partially delegated to AI, reflecting the understanding-gated developer.

5 Understanding AI Reliance through Personas

To address our second research question (RQ2) on the implications of AI reliance, we examine how well participants of the different AIRELI personas performed on the tasks overall, and then take a closer look at how these personas and their characteristics affect behavior and comprehension across the key stages of code change tasks.

5.1 Task Performance with Persona Lens

Across conditions, 10 of 21 participants solved the first task, with 4 of 10 in the No AI group and 6 of 11 in the AI group. Only 3 participants also solved the second transfer task, 1 from the AI group and 2 from the No AI group. Comprehension scores were comparable between conditions (No AI: 0.62 ± 0.26; AI: 0.59 ± 0.23). Taken together, initial results on task performance and comprehension did not reveal major differences between the two conditions. However, taking a look through the **persona lens shows a more nuanced picture**, as elaborated on below and summarized in Figure 4.

AI-steered developers thrive with access to AI, but are completely lost without it. Across our participants, AI-Steered developers (N=8) thrived in implementing task T_1, when given access to AI. Four out of five participants with access to AI finished task T_1. They typically let the model produce an implementation plan and then proceeded to copy/paste code into the codebase (e.g., P5, P6, P14, P18). As soon as AI access was revoked or unavailable, none of the AI-Steered participants were able to finish a task (none finished task T_1 in

Table 1: Characteristics for AI reliance analysis. The set is non-exhaustive and non-exclusive, multiple characteristics may apply to a single participant. It captures the most prominent patterns we observed in the raw data.

ID	Characteristic name and description
System Orientation	
SC	**Strong Cartographer** Builds a coherent mental model of the system; navigates purposefully; produces clear architectural sketches and explanations.
DD	**Disoriented Developer** Frequently loses orientation in the codebase; makes edits in the wrong place; shows repeated backtracking to regain context.
AO	**Architecture Overloaded** Reports to be overwhelmed by relationships between components; struggles to trace end to end flow or "what calls what."
AI Usage Style	
Q&A	**Q&A Style Co Piloter** Uses AI as an interactive tutor in small question and answer cycles; integrates suggestions with verification and adaptation.
HAI	**High AI Use** Spends substantial time using AI; converses with it at high frequency during tasks; a large share of activity flows through AI prompts and responses.
AWF	**AI Withdrawal Frustration** Reports frustration and discouragement when AI is unavailable; momentum depends on continued access to assistance.
IAI	**Instant AI Developer** Delegates implementation to AI immediately without first building understanding; execution is largely copy and paste with shallow verification.
Demographics	
NOV	**Novice** Gaps in the specific technology stack impede progress despite reasonable overall plans or intentions.
EXP	**Expert** Autonomous, fast, accurate; demonstrates deep system understanding; uses assistance strategically rather than dependently.

Table 2: AIRELI Personas, their typical characteristics and reliance on AI (AI) or human expertise (HE) for the main code change stages: Understanding, Planning, Execution.

Persona description and typical characteristics	U	P	E
Self-Sufficient Builds understanding, plans work, and can execute without AI. Uses assistance, if available, as a speed up rather than a crutch. EXP SC	HE	HE	HE
Understanding-Gated All work passes through an understanding gate: The developer verifies plans and code before adoption. Typically builds a mental map first, then relies on AI for implementation specifics during execution. SC Q&A NOV AO AWF	HE	HE	AI
AI-Steered The developer is 'in the passenger seat' as the AI is in charge. It dictates the next step and the developer follows. Verification is deferred or shallow and understanding remains fragile. IAI HAI DD AO AWF NOV	AI	AI	AI

Persona / Condition	Ability to Implement Code Change Task 1		High Code Change and Codebase Comprehension		Ability to Implement Code Change Transfer Task 2	
	AI	No AI	AI	No AI	AI	No AI
Self Sufficient	✓	✓	✓	✓	✓	✓
Understanding Gated	~	~	✓	✓	×	×
AI Steered	✓	×	×	×	×	×

Figure 4: Implementation and comprehension task outcomes by Persona and AI condition (✓: success; ~ : partial success; × : failure).

the No AI condition, and none finished transfer task T_2 when no external help, i.e., no AI, was allowed.

AI-steered developers lack comprehension of their change and the codebase. Beyond execution, most AI-steered participants did not form an accurate mental model of the system. They achieved low comprehension scores (0.36 ± 0.18 on a scale from 0.0 to 1.0), approximately half of what understanding-gated and self-sufficient participants achieved. The architecture sketches further illustrate these differences: seven out of eight AI-steered participants produced incomplete or incorrectly connected system diagrams. The majority scored 0 points for correct relationship mapping and only partial points for correct component identification. For those that successfully completed the first task, none were able to complete the second transfer task successfully, where no external AI assistance was allowed. This lack of comprehension and limited capacity for independent implementation was also reflected in their self-assessed confidence. When asked how confident they would feel performing a similar change in the codebase after completing task T_1, AI-steered participants reported an average confidence of 1.88 ± 0.83 on a five-point scale, the lowest across all participant groups.

Understanding-gated developers emphasize theoretical comprehension but struggle with execution. Understanding-gated developers

(N=10) achieved comprehension levels comparable to their self-sufficient peers. They demonstrated strong architectural and conceptual understanding, with most diagram sketches receiving full scores and consistently solid answers on comprehension questions, yielding an average comprehension score of 0.76 ± 0.12. However, their execution performance lagged behind self-sufficient participants: only three completed Task T_1, and one of those required substantial time (96 min). Almost none completed the transfer task T_2 (participant P1 nearly finished by implementing the login functionality, but ran out of time while configuring access tokens after a successful login). In general, understanding-gated developers followed a map-first approach: they read the code, formed their own mental model, and used AI primarily as a tutor (Q&A, debugging, API/stack nudges), reflecting on and evaluating suggestions rather than pasting them verbatim. This strategy increased overall system understanding but had two consequences for execution-specific issues, such as handling of asynchronous responses, or SQLite wrapper specifics: *(i)* with AI, many lost time in iterative debugging and evaluation loops compared to the immediate copy-paste behavior of AI-steered developers; and *(ii)* without AI, several were blocked by limited execution fluency in the stack despite their strong architectural grasp.

Self-sufficient developers are champions across all measures. Self-sufficient developers (N = 3) excelled regardless of conditions (AI or No AI). All completed Task T_1 with the fastest times (52.0 ± 9.5 minutes) and were the only group to successfully finish the transfer task T_2. Notably, T_2 was completed quickly (23.7 ± 5.9 minutes), demonstrating strong architectural understanding and transfer of knowledge from the preceding change task T_1. Additionally, their comprehension scores were similarly high (0.73 ± 0.10), and they reported the highest confidence across all participants (4.67 ± 0.58 for both tasks). This group was the only one that demonstrated consistent implementation success, deep comprehension, and effective transfer to a new code change.

5.2 Mapping Persona Characteristics to Stages

Breaking the task into three key stages—system understanding, change plan formulation, and change plan execution—provides further insights into how AI reliance, as embodied by the three personas and their characteristics, shapes developer behavior and performance. Figure 5 presents a conceptual overview of these observations through the lens of persona characteristics across the main stages.

System understanding. Understanding the system emerged as a foundational step that set the tone for the entire task session. Prior expertise with the programming language and technology stack influenced how fast participants formed an initial mental model before engaging with codebase specifics. Experts (**EXP**) progressed quickly through language semantics and common idioms, whereas unfamiliar novices (**NOV**) frequently turned to the AI assistant to make sense of what they were seeing before delving into the details. In the AI condition, we often saw Immediate AI Invocation (**IAI**) by novices, where participants' attention shifted to the assistant before they examined the repository. In our data, **IAI** was an early and strong indicator of AI over-reliance associated with the *AI-steered* persona. We also observed Architecture Overwhelm (**AO**) during codebase exploration. While **AO** by itself was not conclusive, since some *understanding-gated* participants also exhibited it, its persistence throughout the session tended to reinforce AI reliance, often carrying forward disorientation (**DD**) into the change plan formulation stage.

Change plan formulation. Since the task required code changes, participants typically formulated a plan after developing an initial understanding of the codebase. When participants demonstrated strong architectural comprehension (**SC**), they generally transitioned into execution with deliberate and purposeful code changes, for instance, connecting key components or implementing core functions. In contrast, when architectural understanding was limited (**DD**), participants often deferred planning to the AI, or, if the AI was unavailable, voiced AI Withdrawal Frustration (**AWF**). Several *understanding-gated* participants interacted with the AI in a Question–Answer (**Q&A**) style to refine or validate their own plans rather than to offload the planning process entirely. In our analysis, this characteristic is considered reliance-neutral, as it reflects an internal evaluative dialogue aimed at improving or confirming a plan, engaging the AI as a collaborator to critique a proposal rather than blindly following its output.

Change plan execution. During the execution stage, we observed high AI use (**HAI**), particularly through direct copy-pasting of suggested solutions. In the No AI condition, participants again displayed frustration (**AWF**) during debugging when they encountered errors they could not resolve on their own, often reporting a longing for AI in the qualitative experience questionnaire. Notably, intensive **Q&A** -style debugging among some *understanding-gated* participants rarely converged on a working solution. While *understanding-gated* participants demonstrated relatively expertise-centric behavior in the first two stages, execution revealed a stronger pull toward AI reliance, manifesting as **HAI** in the AI condition and as **AWF** in the No AI condition. Only the *self-sufficient* participants remained expertise-centric during execution, and all of them successfully completed the task.

Considering additional stages. We focused on the most relevant stages of the code change process, as informed by our observations and collected data, while acknowledging that the programming task process map in Figure 5 represents a simplified model that could be further extended. For instance, task understanding could have been considered as an initial stage. However, we deemed it less relevant in this context because the tasks were intentionally designed to be familiar and unambiguous. Most developers have encountered login and registration workflows, and many participants had recently implemented similar functionality in their software engineering course. Self-reports confirmed generally high task clarity (M = 3.62, SD = 1.02, n = 21). Notably, the lowest ratings ("Slightly," value 2) exclusively stem from participants who exhibited the **IAI** characteristic, suggesting that they may have outsourced even the initial task understanding by immediately pasting the instructions to the AI assistant. Beyond these considerations, code change verification and testing could also be conceptualized as separate stages. However, given our study setup and time constraints, we did not expect

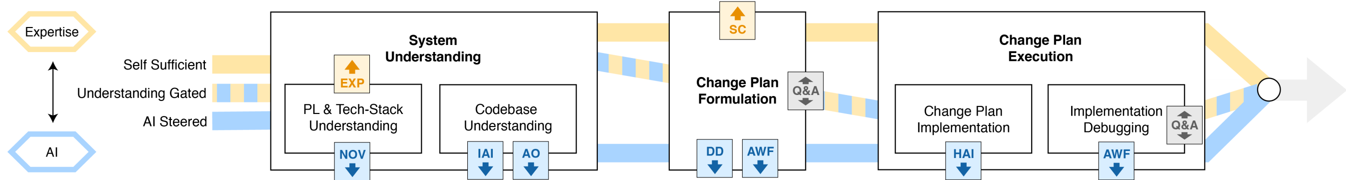

Figure 5: Conceptual integration of personas and characteristics across the main stages of a programming task. Human Expertise (top) and AI (bottom) represent two opposing gravitational forces on the spectrum of AI reliance. Each characteristic can be mapped to one stage or sub-stage, with color and arrow direction indicating its relative contribution towards self-sufficient human expertise or AI reliance. Persona behavior is represented by colored lanes that illustrate the degree of gravitation towards either force across the task stages.

participants to engage in explicit verification, and testing in itself could reasonably be regarded as an additional task.

Persona characteristics, stages, and outsourcing. Examining developer behavior through the lens of persona characteristics and decomposing the change task into three stages enables a **more nuanced understanding of AI reliance and the outsourcing of comprehension and capability to the AI**. We define outsourcing as a shift of the cognitive work required for a given stage or step of the change task from the human developer to the AI assistant, for instance, when a developer is asking the AI for steps to complete a task rather than devising a plan themselves. Our observations suggest that characteristics such as Intermediate AI (IAI) and Architecture Overwhelm (AO) during the system understanding stage can foreshadow patterns of AI reliance, whereas Strong Cartographer (SC) supports self-sufficiency in later stages. At the same time, dialog-style AI use that preserves human evaluation and adaptation, such as seen for the Q&A (Q&A) characteristic that involves asking, evaluating, and adapting, does not constitute outsourcing. In these cases, the AI assistant acts as a sounding board that supplements the developer's own planning and debugging, with the human remaining the primary author of decisions and the change plan.

6 Discussion

Our results provide evidence that viewing developer behavior through the lens of AIRELI personas and the associated characteristics yields a more nuanced understanding of how developers rely on AI across the understanding, planning, and execution stages of change tasks. This perspective reveals patterns of cognitive outsourcing and human-AI collaboration across stages that shape developers' comprehension and their capabilities, patterns that stay hidden when comparing AI and No AI conditions alone. In the following we discuss the necessity of the lenses for program comprehension experiments and more broadly the potential impact of AI reliance for junior developers.

6.1 Necessity for AI Reliance Considerations in Program Comprehension Experiments

Without an explicit lens on AI reliance, results from code comprehension experiments become noisy and might lead to misleading interpretations of results. We experienced this challenge firsthand in our data collection study, where roughly half of the participants completed the first task both in the AI and No AI conditions, and performed similarly on the comprehension questionnaires. Taken at face value, such outcomes might suggest that AI usage has barely any effect on task completion or code comprehension. Even accounting for self-reported general AI usage and reliance in a demographics question did not change this picture. However, when analyzed through a more comprehensive AI reliance lens, the data revealed more diverse and distinct patterns of capability and understanding that were otherwise obscured.

When comprehension is outsourced, there is little human comprehension left to examine. Modern AI assistants can propose plans, generate code, and steer debugging. If participants delegate these steps, the construct that a comprehension study aims to measure is partially or fully outsourced to the assistant. Task completion then reflects successful delegation and AI model accuracy rather than human understanding. The usual proxies for comprehension, such as success, time, or error counts, become unreliable. Experiments must therefore distinguish pathways that preserve human authorship of understanding from pathways that substitute it. In our paper, we examine this distinction through characteristics and personas to indicate and separate AI reliance types based on their ability to understand, plan, and execute.

Traditional control conditions become fragile once AI is integral to programming. A common approach to studying the effects of AI on programming ability and comprehension is to compare an AI-assisted 'treatment' group with a No AI baseline. However, when AI assistance has already become an integral part of everyday development, removing it introduces unexpected challenges. In our study, many participants, most of whom were students with access to AI and reported high AI usage, already internalized AI assistance into their workflows and are unable to proceed without it. For some participants, AI reliance occurred as early as during system understanding or change plan formulation, but for most, at the latest when relying on it during execution. Future studies could, in principle, treat the removal of AI as the experimental condition. However, this approach would still require a baseline group of participants who do not rely on AI for comparison, for instance, to confirm that non-reliant developers would even be able to complete the task independently and to clearly isolate AI effects. One possible baseline could involve recruiting more senior developers who did not have access to AI during their education and have since chosen to work without it, or at least without being reliant on it. While such

study designs may be limited to narrow participant populations or create somewhat artificial experimental contexts, conducting them soon may be worthwhile, as the pool of such non-reliant baseline developers is likely to shrink over time if AI use continues to expand.

Reliance aware experiment designs and reporting. To make comprehension experiments suitable for an AI-assisted development context, we recommend analysis approaches that foreground AI reliance rather than averaging it away. Future work should explore reliance-aware methods for measuring human program comprehension that do not conflate it with AI substitution. Based on our observations, promising starting points include asking participants to sketch task-relevant system architectures, analyzing code authorship through chat logs, and potentially using small transfer-task probes. Since we might be able to operationalize most of our characteristics, another direction for future work is to automatically infer a participant's AI reliance persona from these computed characteristics.

6.2 Implications of AI Reliance: The Junior Developer AI Fallacy

In this study, we observed that many developers outsourced substantial portions of their work to AI assistants, effectively substituting human expertise with the output of AI models. A surprising observation was that many AI-steered developers still succeeded in completing the first task quickly *without* comprehending most of the codebase or even their own changes. This observation raises central questions: if tasks get done, why should we care about comprehension? Is it acceptable to rely on AI, and how much reliance is appropriate? Looking at comprehension as a core part of learning [1, 10] and as a central, time-consuming activity in professional software work [7, 42, 49], it becomes clear why the degree of reliance on AI could matter for future generations of developers. When comprehension is outsourced, learning can plateau, and transferable expertise may not develop. Classic human–automation research further warns that overreliance can lead to out-of-the-loop effects and degraded intervention skills [17, 35, 47]. We emphasize that the following are *potential implications* stemming from our persona-based observations, not confirmed long-term effects. Longitudinal studies are needed to confirm or reject the fallacy hypothesis. However, raising this as a potential concern now is timely so that we can begin to anticipate and mitigate unwanted effects rather than waiting until it is too late.

We tentatively refer to this risk as the *Junior Developer AI Fallacy*: the assumption that early productivity gains from strong assistant usage among newcomers imply sustained advantage and reduced need for junior talent. AI-steered juniors can appear ahead because they delegate planning and execution of easy, entry-level tasks to the AI, moving from prompt to successful implementation in quick turnaround times. As task complexity rises with seniority, this advantage can quickly erode relative to initially understanding-gated peers. Early gains give way to stagnation bounded by the model's capabilities and can even turn into productivity decline as task complexity grows while capabilities stay the same. The underlying mechanism is **comprehension debt**: outsourcing understanding and plan formulation can build up a deficit that seems harmless on simple work but could become costly as complexity grows. Two forces drive this shift: *(i)* AI suggestion quality often deteriorates as task complexity and project maturity increase [34], and *(ii)* reliance patterns potentially limit learning, such that when plan authorship and debugging are outsourced, knowledge transfer to new tasks is limited and each new challenge starts from scratch.

Implications for juniors: going through the understanding-gate towards a path for long-term success. For newcomers, the dilemma is clear: ignoring assistance reduces short-term efficiency, yet over-reliance prevents long-term growth and benefits. A reliance aware path keeps authorship of understanding while using the assistant only through an understanding gate: practical anchors could include *(i)* creating a habit of building a map like understanding of task relevant parts of the system first, *(ii)* formulating a change plan draft before consulting AI, or when initially drafting with AI, keeping the understanding gate alive by critically evaluating and justifying each step, and *(iii)* delayed delegation where code is generated only for well bounded substeps after the plan exists and with explicit verification criteria. These anchors align with our personas: self-sufficient and understanding-gated developers grow; AI-steered developers plateau unless the workflow is reoriented toward authorship.

Implications for managers and teams. The fallacy also tempts managers to conclude that fewer juniors are needed because assistants can produce the bulk of routine code and solve entry-level tasks. Even if this holds for near-term work, it undermines the pathway that develops future senior developers who can handle the complex, ambiguous work where assistants are least effective [34]. Teams can mitigate by making reliance visible and coachable. Useful practices could include *(i)* capturing code origin through AI chat logs, paste events, and diffs to see authorship, *(ii)* using AI reliance personas during onboarding and growth conversations to target coaching, *(iii)* structuring reviews that require developers to explain their change plan and show how suggestions were verified. The goal is not to ban AI assistance but to ensure that juniors progress from AI-steered toward understanding-gated and ultimately self-sufficient developers.

7 Threats to Validity

External validity. Our participants were primarily students, which limits generalizability to professional engineers with deeper, domain-specific experience or established team practices. The tasks (implementing registration and login) in a TypeScript monorepo are realistic but still specific; reliance patterns may differ for other stacks or legacy codebases. We studied a chat-based assistant with inline code-completion disabled and allowed ordinary web resources; other ways to interact with AI assistance (IDE-integrated copilots, retrieval-augmented chat, organization-specific tooling and tests) might shift how reliance manifests. Finally, our single-session lab setting emphasizes near-term execution and a single transfer step; it does not capture how long-term learning is affected, how accumulation/repayment of "comprehension debt" could unfold, requiring further exploration by future work.

Internal validity. This observational study aims to surface emerging AI-reliance patterns rather than make causal claims. We limited confounds via standardized onboarding (identical briefing and setup help), identical materials (same monorepo and task acceptance checks), a controlled environment (participants used their own machines which is ecologically realistic but heterogeneous; code-completion disabled; AI access restricted to a provided ChatGPT account to capture comparable logs), and consistent instrumentation (screen recordings, time-stamped transcripts, uniform questionnaires). Given these limitations, we report descriptive patterns; our analyses indicate that prior expertise is a major factor in AI reliance, so future confirmatory work should explicitly account for it and examine whether different modes of AI assistance shape users' reliance profiles.

Construct validity. Code comprehension metrics such as task success or time-to-completion may reflect AI assistant effectiveness rather than human understanding. We mitigated this by pairing an AI-assisted task with a follow-up unaided transfer task, and by triangulating reliance with a comprehension questionnaire (rubric-scored sketch of the task-relevant architecture, and K-Prim multiple choice questions). Still, in a no AI condition, weak performance can be ambiguous because it can signal limited skill or habitual overreliance revealed only when assistance is removed; when AI is available, strong performance may reflect the assistant substituting for missing expertise. While in most cases reliance can clearly be determined, in these more difficult cases we triangulate reliance by using multiple signals, including self reports such as "I can't do anything without AI" or "I don't know where to start without AI", general AI usage reports, and behavioral traces in logs and recordings.

8 Conclusion

We conducted a controlled study with 21 participants using a realistic junior developer task setting to examine developer behaviors and reliance on AI across system understanding, plan formulation, and execution. Based on an analysis of the quantitative and qualitative data collected, we derived three AI reliance personas and nine associated characteristics. Using this persona-based lens revealed that aggregate comparisons between AI and No AI conditions masked key behavioral and outcome differences in program comprehension and the ability to complete tasks that emerged only after grouping participants by AI reliance personas. Self-sufficient developers relied on their own expertise across all three stages: building a sound mental model, devising a plan, and implementing it. Understanding-gated developers remain in charge of system understanding and formulating a change plan but frequently rely on AI during execution and debugging, achieving solid comprehension while execution remains their main point of AI reliance. AI-steered developers outsourced most aspects of the task to the AI assistant, including system understanding, plan formulation, and implementation, which can still yield successful task completion but leads to shallow comprehension. We contribute a compact taxonomy of three personas that separates human-authored understanding and capability from assistant substitution, offer reliance-aware design recommendations, and call for confirmatory studies to generalize and refine these findings across diverse tasks and populations.

Acknowledgments

We thank all study participants for taking part in our study, which made this work possible. This material is based upon work supported by the National Science Foundation under grants CCF-2210812 and CCF-2210813. It is further supported by the Federal Institute of Education, Science and Technology of Rio Grande do Sul (IFRS) and by the Ministry of Science, Technology, and Innovation of Brazil (Law 8.248 from Oct 23, 1991), within the scope of PPI-SOFTEX, coordinated by Softex, and published in the Residência em TIC 02 - Aditivo, Official Gazette 01245.012095/2020-56. We acknowledge the support of the Natural Sciences and Engineering Research Council of Canada (NSERC), application IDs 594057-2024 and 606783/2025. Finally, we thank the University of Zurich and the Digital Society Initiative for partially financing this work.

References

[1] 2018. *How People Learn II: Learners, Contexts, and Cultures.* National Academies Press, Washington, D.C. doi:10.17226/24783
[2] Tarek Alakmeh, Nathan Anderson, Victoria Jackson, Gustavo Vaz Pereira, Umutcan Akirmak, Amanda Estey, Rafael Prikladnicki, André van der Hoek, Margaret-Anne Storey, and Thomas Fritz. 2026. AIRELI Replication Package: Grasping AI Reliance in Program Comprehension and Coding through the AIRELI Persona Taxonomy. doi:10.5281/zenodo.18328348
[3] Shraddha Barke, Michael B. James, and Nadia Polikarpova. 2023. Grounded Copilot: How Programmers Interact with Code-Generating Models. *Replication Package for Article: "Grounded Copilot: How Programmers Interact with Code-Generating Models"* 7, OOPSLA1 (April 2023), 78:85–78:111. doi:10.1145/3586030
[4] Joel Becker, Nate Rush, Beth Barnes, and David Rein. 2025. Measuring the Impact of Early-2025 AI on Experienced Open-Source Developer Productivity. (July 2025).
[5] Blazity. 2024. Next.js enterprise-grade storefront for high-performance e-commerce. https://github.com/Blazity/enterprise-commerce.
[6] Virginia Braun and Victoria Clarke. 2006. Using thematic analysis in psychology. *Qualitative Research in Psychology* 3, 2 (2006), 77–101. doi:10.1191/1478088706qp063oa
[7] Ruven Brooks. 1983. Towards a theory of the comprehension of computer programs. *International Journal of Man-Machine Studies* 18, 6 (June 1983), 543–554. doi:10.1016/S0020-7373(83)80031-5
[8] Adam Brown, Sarah D'Angelo, Ambar Murillo, Ciera Jaspan, and Collin Green. 2024. Identifying the Factors That Influence Trust in AI Code Completion. In *Proceedings of the 1st ACM International Conference on AI-Powered Software (AIware 2024).* Association for Computing Machinery, New York, NY, USA, 1–9. doi:10.1145/3664646.3664757
[9] Tuan-Dung Bui, Thanh Trong Vu, Thu-Trang Nguyen, Son Nguyen, and Hieu Dinh Vo. 2025. Correctness assessment of code generated by Large Language Models using internal representations. *Journal of Systems and Software* 230 (Dec. 2025), 112570. doi:10.1016/j.jss.2025.112570
[10] Michelene T. H. Chi, Nicholas De Leeuw, Mei-Hung Chiu, and Christian Lavancher. 1994. Eliciting self-explanations improves understanding. *Cognitive Science* 18, 3 (July 1994), 439–477. doi:10.1016/0364-0213(94)90016-7
[11] Rudrajit Choudhuri, Dylan Liu, Igor Steinmacher, Marco Gerosa, and Anita Sarma. 2023. How Far Are We? The Triumphs and Trials of Generative AI in Learning Software Engineering. doi:10.48550/arXiv.2312.11719 arXiv:2312.11719 [cs].
[12] Autumn Clark, Daniel Igbokwe, Samantha Ross, and Minhaz F. Zibran. 2024. A Quantitative Analysis of Quality and Consistency in AI-generated Code. In *2024 7th International Conference on Software and System Engineering (ICoSSE).* 37–41. doi:10.1109/ICoSSE62619.2024.00014
[13] Vincenzo Corso, Leonardo Mariani, Daniela Micucci, and Oliviero Riganelli. 2024. Generating Java Methods: An Empirical Assessment of Four AI-Based Code Assistants. In *Proceedings of the 32nd IEEE/ACM International Conference on Program Comprehension (ICPC '24).* Association for Computing Machinery, New York, NY, USA, 13–23. doi:10.1145/3643916.3644402
[14] Zheyuan (Kevin) 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. doi:10.2139/ssrn.4945566
[15] Nicole Davila, Igor Wiese, Igor Steinmacher, Lucas Lucio da Silva, Andre Kawamoto, Gilson Jose Peres Favaro, and Ingrid Nunes. 2024. An Industry Case Study on Adoption of AI-based Programming Assistants. In *Proceedings of the 46th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP '24).* Association for Computing Machinery, New York, NY, USA, 92–102. doi:10.1145/3639477.3643648

[16] Tasneem Muhammed Eltabakh, Nada Nabil Soudi, and Doaa Shawky. 2024. Quality of AI-Generated vs. Human-Generated Code. In *2024 34th International Conference on Computer Theory and Applications (ICCTA)*. 200–205. doi:10.1109/ICCTA64612.2024.10974782 ISSN: 2770-6575.

[17] Mica R. Endsley and Esin O. Kiris. 1995. The Out-of-the-Loop Performance Problem and Level of Control in Automation. *Human Factors* 37, 2 (June 1995), 381–394. doi:10.1518/001872095779064555 Publisher: SAGE Publications Inc.

[18] Thomas Fritz, David C. Shepherd, Katja Kevic, Will Snipes, and Christoph Bräunlich. 2014. Developers' code context models for change tasks. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Hong Kong, China) *(FSE 2014)*. Association for Computing Machinery, New York, NY, USA, 7–18. doi:10.1145/2635868.2635905

[19] Huizi Hao, Kazi Amit Hasan, Hong Qin, Marcos Macedo, Yuan Tian, Steven H. H. Ding, and Ahmed E. Hassan. 2024. An empirical study on developers' shared conversations with ChatGPT in GitHub pull requests and issues. *Empirical Softw. Engg.* 29, 6 (Sept. 2024). doi:10.1007/s10664-024-10540-x

[20] Cristina Improta. 2023. Poisoning Programs by Un-Repairing Code: Security Concerns of AI-generated Code. In *2023 IEEE 34th International Symposium on Software Reliability Engineering Workshops (ISSREW)*. 128–131. doi:10.1109/ISSREW60843.2023.00060

[21] Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2024. SWE-bench: Can Language Models Resolve Real-World GitHub Issues? doi:10.48550/arXiv.2310.06770 arXiv:2310.06770 [cs].

[22] Samia Kabir, David N. Udo-Imeh, Bonan Kou, and Tianyi Zhang. 2024. Is Stack Overflow Obsolete? An Empirical Study of the Characteristics of ChatGPT Answers to Stack Overflow Questions. In *Proceedings of the 2024 CHI Conference on Human Factors in Computing Systems (CHI '24)*. Association for Computing Machinery, New York, NY, USA, 1–17. doi:10.1145/3613904.3642596

[23] Ranim Khojah, Mazen Mohamad, Philipp Leitner, and Francisco Gomes de Oliveira Neto. 2024. Beyond Code Generation: An Observational Study of ChatGPT Usage in Software Engineering Practice. *Proc. ACM Softw. Eng.* 1, FSE (July 2024), 81:1819–81:1840. doi:10.1145/3660788

[24] Thomas D. LaToza, Gina Venolia, and Robert DeLine. 2006. Maintaining mental models: a study of developer work habits. In *Proceedings of the 28th International Conference on Software Engineering* (Shanghai, China) *(ICSE '06)*. Association for Computing Machinery, New York, NY, USA, 492–501. doi:10.1145/1134285.1134355

[25] Teemu Lehtinen, Charles Koutcheme, and Arto Hellas. 2024. Let's Ask AI About Their Programs: Exploring ChatGPT's Answers To Program Comprehension Questions. In *Proceedings of the 46th International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET '24)*. Association for Computing Machinery, New York, NY, USA, 221–232. doi:10.1145/3639474.3640058

[26] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2023. Is your code generated by ChatGPT really correct? rigorous evaluation of large language models for code generation. In *Proceedings of the 37th International Conference on Neural Information Processing Systems (NIPS '23)*. Curran Associates Inc., Red Hook, NY, USA, 21558–21572.

[27] 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 (June 2024), 116:1–116:26. doi:10.1145/3643674

[28] Ran Mo, Dongyu Wang, Wenjing Zhan, Yingjie Jiang, Yepeng Wang, Yuqi Zhao, Zengyang Li, and Yutao Ma. 2025. Assessing and Analyzing the Correctness of GitHub Copilot's Code Suggestions. *ACM Trans. Softw. Eng. Methodol.* 34, 7 (Aug. 2025), 194:1–194:32. doi:10.1145/3715108

[29] Hussein Mozannar, Gagan Bansal, Adam Fourney, and Eric Horvitz. 2024. Reading Between the Lines: Modeling User Behavior and Costs in AI-Assisted Programming. In *Proceedings of the 2024 CHI Conference on Human Factors in Computing Systems (CHI '24)*. Association for Computing Machinery, New York, NY, USA, 1–16. doi:10.1145/3613904.3641936

[30] Daye Nam, Andrew Macvean, Vincent Hellendoorn, Bogdan Vasilescu, and Brad Myers. 2024. Using an LLM to Help With Code Understanding. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (ICSE '24)*. Association for Computing Machinery, New York, NY, USA, 1–13. doi:10.1145/3597503.3639187

[31] Claudia Negri-Ribalta, Rémi Geraud-Stewart, Anastasia Sergeeva, and Gabriele Lenzini. 2024. A systematic literature review on the impact of AI models on the security of code generation. *Frontiers in Big Data* 7 (May 2024). doi:10.3389/fdata.2024.1386720 Publisher: Frontiers.

[32] Sydney Nguyen, Hannah McLean Babe, Yangtian Zi, Arjun Guha, Carolyn Jane Anderson, and Molly Q Feldman. 2024. How Beginning Programmers and Code LLMs (Mis)read Each Other. In *Proceedings of the 2024 CHI Conference on Human Factors in Computing Systems (CHI '24)*. Association for Computing Machinery, New York, NY, USA, 1–26. doi:10.1145/3613904.3642706

[33] Delano Oliveira, Reydne Bruno, Fernanda Madeiral, and Fernando Castor. 2020. Evaluating Code Readability and Legibility: An Examination of Human-centric Studies. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 348–359. doi:10.1109/ICSME46990.2020.00041

[34] Elise Paradis, Kate Grey, Quinn Madison, Daye Nam, Andrew Macvean, Vahid Meimand, Nan Zhang, Ben Ferrari-Church, and Satish Chandra. 2024. How much does AI impact development speed? An enterprise-based randomized controlled trial. doi:10.48550/arXiv.2410.12944 arXiv:2410.12944 [cs].

[35] R. Parasuraman, T.B. Sheridan, and C.D. Wickens. 2000. A model for types and levels of human interaction with automation. *IEEE Transactions on Systems, Man, and Cybernetics - Part A: Systems and Humans* 30, 3 (May 2000), 286–297. doi:10.1109/3468.844354

[36] Hammond Pearce, Baleegh Ahmad, Benjamin Tan, Brendan Dolan-Gavitt, and Ramesh Karri. 2025. Asleep at the Keyboard? Assessing the Security of GitHub Copilot's Code Contributions. *Commun. ACM* 68, 2 (Jan. 2025), 96–105. doi:10.1145/3610721

[37] Sida Peng, Eirini Kalliamvakou, Peter Cihon, and Mert Demirer. 2023. The Impact of AI on Developer Productivity: Evidence from GitHub Copilot. doi:10.48550/arXiv.2302.06590 arXiv:2302.06590 [cs].

[38] Nancy Pennington. 1987. Stimulus structures and mental representations in expert comprehension of computer programs. *Cognitive Psychology* 19, 3 (1987), 295–341. doi:10.1016/0010-0285(87)90007-7

[39] 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 (Nov. 2023), 4:1–4:31. doi:10.1145/3617367

[40] 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 (ICER '24, Vol. 1)*. Association for Computing Machinery, New York, NY, USA, 469–486. doi:10.1145/3632620.3671116

[41] Anshul Shah, Thomas Rexin, Anya Chernova, Gonzalo Allen-Perez, William G. Griswold, and Adalbert Gerald Soosai Raj. 2025. Needles in a Haystack: Student Struggles with Working on Large Code Bases. In *Proceedings of the 2025 ACM Conference on International Computing Education Research V.1 (ICER '25)*. Association for Computing Machinery, New York, NY, USA, 27–40. doi:10.1145/3702652.3744218

[42] M.-A. Storey. 2005. Theories, methods and tools in program comprehension: past, present and future. In *13th International Workshop on Program Comprehension (IWPC'05)*. 181–191. doi:10.1109/WPC.2005.38 ISSN: 1092-8138.

[43] Ningzhi Tang, Meng Chen, Zheng Ning, Aakash Bansal, Yu Huang, Collin McMillan, and Toby Jia-Jun Li. 2024. A Study on Developer Behaviors for Validating and Repairing LLM-Generated Code Using Eye Tracking and IDE Actions. doi:10.48550/arXiv.2405.16081 arXiv:2405.16081 [cs].

[44] Priyan Vaithilingam, Tianyi Zhang, and Elena L. Glassman. 2022. Expectation vs. Experience: Evaluating the Usability of Code Generation Tools Powered by Large Language Models. In *Extended Abstracts of the 2022 CHI Conference on Human Factors in Computing Systems (CHI EA '22)*. Association for Computing Machinery, New York, NY, USA, 1–7. doi:10.1145/3491101.3519665

[45] A. von Mayrhauser and A. M. Vans. 1994. Comprehension processes during large scale maintenance. In *Proceedings of the 16th International Conference on Software Engineering* (Sorrento, Italy) *(ICSE '94)*. IEEE Computer Society Press, Washington, DC, USA, 39–48.

[46] Ruotong Wang, Ruijia Cheng, Denae Ford, and Thomas Zimmermann. 2024. Investigating and Designing for Trust in AI-powered Code Generation Tools. In *Proceedings of the 2024 ACM Conference on Fairness, Accountability, and Transparency (FAccT '24)*. Association for Computing Machinery, New York, NY, USA, 1475–1493. doi:10.1145/3630106.3658984

[47] C. D. Wickens. 1995. Designing for Situation Awareness and Trust in Automation. *IFAC Proceedings Volumes* 28, 23 (Sept. 1995), 365–370. doi:10.1016/S1474-6670(17)46646-8

[48] Marvin Wyrich, Justus Bogner, and Stefan Wagner. 2022. 40 Years of Designing Code Comprehension Experiments: A Systematic Mapping Study. doi:10.48550/arXiv.2206.11102 arXiv:2206.11102 [cs].

[49] Xin Xia, Lingfeng Bao, David Lo, Zhenchang Xing, Ahmed E. Hassan, and Shanping Li. 2018. Measuring program comprehension: a large-scale field study with professionals. In *Proceedings of the 40th International Conference on Software Engineering (ICSE '18)*. Association for Computing Machinery, New York, NY, USA, 584. doi:10.1145/3180155.3182538

[50] Yuankai Xue, Hanlin Chen, Gina R. Bai, Robert Tairas, and Yu Huang. 2024. Does ChatGPT Help With Introductory Programming?An Experiment of Students Using ChatGPT in CS1. In *Proceedings of the 46th International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET '24)*. Association for Computing Machinery, New York, NY, USA, 331–341. doi:10.1145/3639474.3640076

[51] Litao Yan, Alyssa Hwang, Zhiyuan Wu, and Andrew Head. 2024. Ivie: Lightweight Anchored Explanations of Just-Generated Code. In *Proceedings of the 2024 CHI Conference on Human Factors in Computing Systems (CHI '24)*. Association for

Computing Machinery, New York, NY, USA, 1–15. doi:10.1145/3613904.3642239
[52] Yangtian Zi, Luisa Li, Arjun Guha, Carolyn Anderson, and Molly Q Feldman. 2025. "I Would Have Written My Code Differently': Beginners Struggle to Understand LLM-Generated Code. In *Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering (FSE Companion '25)*. Association for Computing Machinery, New York, NY, USA, 1479–1488. doi:10.1145/3696630.3731663
[53] Albert Ziegler, Eirini Kalliamvakou, X. Alice Li, Andrew Rice, Devon Rifkin, Shawn Simister, Ganesh Sittampalam, and Edward Aftandilian. 2024. Measuring GitHub Copilot's Impact on Productivity. *Commun. ACM* 67, 3 (Feb. 2024), 54–63. doi:10.1145/3633453