Context-Specific Instruction: A Longitudinal Study on Debugging Skill Acquisition and Retention for Novice Programmers

ZIYI ZHANG, DEVJEET ROY, and VENERA ARNAOUDOVA, Washington State University, USA

Objectives: Bug localization is a critical skill for programmers, but novices often struggle to develop systematic approaches. While prior work has explored abstract guidelines and general concrete steps, the impact of context-specific instruction remains unexplored. This study introduces and evaluates a novel context-specific instruction approach, comparing it with existing instruction methods to understand how different specificity levels affect novices' debugging (localization phase) skill acquisition and retention.

Participants: 44 undergraduate Computer Science students participated in the experiment, with 41 completing all five sessions over eight weeks.

Study Methods: We conducted an eight-week longitudinal study comparing four instruction types: no instructions ((Group 1, referred to as G1 for short), instructions containing abstract guidelines (G2), instructions with concrete steps (G3), and finally, instructions with our proposed context-specific examples that integrate concrete bug localization steps with problem-specific implementation details (G4). Participants (N=44) completed four weekly sessions (S1–S4) measuring short-term skill acquisition and another session (S5) after three weeks to assess long-term retention. Each session included 2-3 debugging tasks to identify the minimal code element containing a seeded logical fault, with performance tracked via correctness (binary success), time-to-completion, and qualitative feedback on cognitive load and strategy adherence.

Findings: Our proposed context-specific instruction (G4) achieved 80% correctness after one session (vs. 20–44% for other groups) and sustained 80% correctness after three weeks, outperforming all groups ($p < 0.05$). G4 achieved stable time-to-completion (13–15 minutes) from the first session onward, while other groups required 2–3 sessions to stabilize (22–27 minutes). These results reveal that time-efficiency plateaus early, whereas correctness improves gradually with practice, highlighting the need for sustained training with novices to better master bug localization skills. Qualitative data highlighted reduced stress levels and higher satisfaction in G4, with participants internalizing strategies through contextual examples.

Conclusion: This study demonstrates that our context-specific instruction approach significantly outperforms traditional abstract guidelines, offering both rapid skill acquisition and robust retention. Even 1–2 sessions yield significant gains, while extended practice optimizes performance. Instructors can implement brief interventions for early improvement or comprehensive programs for optimal and most stabilized outcomes. These findings indicate that integrating contextual examples with abstract principles could bridge theory-practice gaps in bug localization education, offering equitable pathways for novice programmers.

CCS Concepts: • **Social and professional topics** → **Student assessment**; **Computer science education**; • **Applied computing** → **Education**.

Additional Key Words and Phrases: debugging instruction, novice programmers, code comprehension, bug localization, computer science education, longitudinal study, experimental design

Authors' Contact Information: Ziyi Zhang; Devjeet Roy; Venera Arnaoudova, School of Electrical Engineering and Computer Science, Washington State University, Pullman, WA, USA, ziyi.zhang2@wsu.edu, devjeet.roy@wsu.edu, venera.arnaoudova@wsu.edu.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2025 Copyright held by the owner/author(s). Publication rights licensed to ACM.

Manuscript submitted to ACM

ACM Reference Format:
Ziyi Zhang, Devjeet Roy, and Venera Arnaoudova. 2025. Context-Specific Instruction: A Longitudinal Study on Debugging Skill Acquisition and Retention for Novice Programmers. 1, 1 (September 2025), 31 pages.

1 Introduction

Debugging is a fundamental skill in software development [28, 78, 80], yet novice programmers often struggle to develop systematic approaches to locating [55, 63, 76] and resolving code defects [21, 69, 81, 98]. While experienced developers leverage established debugging practices [39, 41], translating these expert strategies into effective instruction remains a significant challenge in computing education [3, 14, 21, 51]. This challenge is exacerbated by the wide variability in pedagogical approaches; educators must choose between abstract guidelines and concrete step-by-step instructions [75], with little empirical evidence to guide decisions about which approach best fosters skill development and long-term retention [11, 81, 87]. The absence of evidence-based instruction frameworks leaves many educators uncertain about how to structure debugging education for optimal learning outcomes [40, 64, 65, 101], which has a particular impact on marginalized learners: as equitable pedagogy requires contextualized support to reduce implicit cognitive loads [46, 56, 66, 86].

Prior work has explored diverse approaches to debugging instruction, yet each approach reveals important limitations. Abstract guidelines [53] demonstrate improvements in self-efficacy but often lack practical applicability, while structured approaches [57] show immediate benefits but unclear long-term impact. Longitudinal studies in programming education [7, 27, 59, 103] further suggest that comprehension skills develop nonlinearly, indicating instruction effectiveness may vary across learning phases.

Despite these advances, critical gaps persist in understanding *(i) how different instruction levels influence the rate and pattern of bug localization skill development, (ii) how durable the acquired debugging skills remain after extended periods without practice, and (iii) how many instructional sessions are needed to achieve stable performance improvements.* Single-session studies [53, 57] risk conflating novelty effects with genuine learning, while overlooking critical patterns in skill retention and performance stabilization—factors essential for designing effective real-world educational interventions [34, 78, 90].

In this paper, to address these gaps, we present an eight-week longitudinal study that introduces and evaluates a novel context-specific instruction approach for debugging. Our study compares four instruction types over multiple sessions: a control group using self-guided strategies (No instruction (G1)), traditional abstract guidelines based on Ko et al.'s framework [53] (Abstract guidelines (G2)), concrete steps following Latoza et al.'s approach [57] (Concrete, context-agnostic (G3)), and our proposed context-specific instruction that uniquely integrates actionable debugging steps with code-specific implementation details (Context-specific examples (G4)). Through analysis combining quantitative performance metrics (task correctness and completion time) with qualitative feedback, we examine 44 novice programmers across five sessions (four weekly sessions S1-S4, and another session conducted after a 3-week gap, i.e., S5) to understand how instruction specificity influences both immediate performance (S1-S4) and long-term skill retention (S5). We examine the bug localization sub-skill within the broader debugging process, as participants were tasked with identifying the minimal code elements that contained a seeded logical fault (i.e., the bug is only contained in one statement) without modifying the code. Thus, our measures could reflect where and how quickly students localized faults rather than diagnosis or repair quality. We therefore interpret effects as improvements in localization, i.e., the "find" phase of debugging, and discuss the implications for end-to-end debugging in the Limitations. Our results show that context-specific instruction could lead to significant short-term improvements towards participants

performance, which indicates it enables rapid skill acquisition (G4: 80% correctness rate and 13 minutes to complete the task after one session, however G1-3 are with 20-44% correctness rate and need 27-31 minutes to finish tasks). In addition, G4 demonstrates the best long term impact comparing to other groups (maintains at 80% and 14 minutes time-to-completion even after a three-week gap). In terms of retention, time efficiency stabilizes within one sessions for all groups, while for correctness rate, only G4 stabilizes after 2 sessions, other groups has more fluctuating trajectories. These findings suggest that while conceptual instructions alone are insufficient, complementing them with contextual examples can effectively bridge the gap between theory and practice in debugging education.

Soundness Our analysis follows a mixed-methods design that contains both quantitative and qualitative analysis: non-parametric tests address small-sample limitations, learning curve analysis [47] models skill progression, and qualitative feedback triangulates results. We also randomize participants' group assignments and code snippets to balance group baselines, while selecting open-source code snippets from GitHub to ensure ecological validity.

Significance This study advances debugging education through five key **contributions**:

(1) We present the first longitudinal study comparing how instruction types impact both short-term skill acquisition and long-term retention in debugging. Our eight-week design (S1–S5) reveals how novices internalize strategies over time, moving beyond single-session evaluations that dominate prior work.
(2) We demonstrate that contextual examples (integrating code-specific details with procedural steps) reduce cognitive load and enable novices to achieve highest correctness and shortest time-to-completion. These examples bridge the theory-practice gap, offering a blueprint for equitable pedagogy that supports marginalized learners lacking prior debugging exposure.
(3) We show that even 1–2 sessions with context-specific instruction yield significant gains, while extended practice (2–3 sessions) optimizes accuracy. This flexibility allows educators to tailor interventions to curricular constraints without sacrificing learning outcomes.
(4) Contrary to intuition, we find that abstract debugging frameworks may be counterproductive for novice programmers, who struggle to translate high-level guidance into concrete actions. This increases cognitive strain without providing sustained benefits. Our findings suggest augmenting theoretical guidelines with context-specific examples, creating a more accessible pathway for novice skill development.
(5) By integrating learning curve principles, we reveal divergent stabilization patterns: time-to-completion plateaus within 1 session with contextual guidance, while correctness improves with one more sessions then stabilizes. This insight enables educators to scaffold instruction dynamically—emphasizing efficiency early and accuracy later.

Data Availability / Replication Package The replication package contains anonymized participant data, analysis scripts, and the full set of instruction texts (for all code snippets) for G2–G4. [1].

Paper organization The rest of the paper is organized as follows. Section Related Works discusses backgrounds and reviews prior work in debugging education and learning curves. Sections Methodology defines our research questions and presents the experimental set up and methodology used to answer those research questions. Section Results presents the results across RQs and discusses the key findings of this study. Section Threats to Validity discusses the threats to validity of this work. Section Conclusion concludes this study and discusses future works.

[1]The replication package, including anonymized data, code, and instruction templates, is available at: https://github.com/anonymoussubmission4papers-sys/Context-Specific-Instruction

2 Methodology

The *objective* of this study is twofold: First, to investigate how different instructions impact novice programmers' **bug localization performance**, i.e., the ability to accurately and efficiently identify the minimal code element containing a seeded fault—in both short-term and long-term; Second, to identify how many training sessions are needed for novices to achieve stable and optimal localization performance. The *perspective* is that of computing education researchers and instructors interested in leveraging longitudinal performance trends and qualitative feedback from novices to improve their **bug localization skill development**, as a foundational component of the broader debugging process.

In this section, we formulate the research questions that guide this study, describe the source code snippets and metrics used to evaluate the performance of participants, and detail the experimental procedure designed to assess the effectiveness of the instruction in multiple sessions. We also detail the data collection and preprocessing methods, as well as the analytical approach employed to address each research question.

2.1 Research Questions

Our investigation is guided by four research questions, designed to understand both short-term and long-term effects of different instruction specificity on bug localization performance.

2.1.1 *RQ1: How do novice' debugging performance trends differ across instruction groups over multiple sessions?*

Previous research suggests that providing systematic **bug localization instructions** (i.e., strategies for the localization phase of debugging) to novice programmers might improve their performance [12, 53, 57, 73, 74]. However, the impact of instruction specificity levels - from abstract guidelines to highly detailed, actionable steps - on learning trajectories over time still remains unexplored. Since learning to debug systematically is a gradual process, and also the rate and pattern of improvement may vary with instruction type, this research question examines how students' performance evolve when trained with four different instruction specificity levels: no instruction, abstract three-step process, detailed step-by-step guidance, and context-specific examples with concrete implementation details.

2.1.2 *RQ2: What is the short-term impact of different instruction specificity levels on novice performance?*

Prior work has explored the effects of abstract [12, 53] or concrete debugging guidance [57, 73, 74] on immediate outcomes, but the comparative effectiveness of instruction specificity levels across repeated sessions remains under-examined. We define *short-term impact* as the effectiveness of instruction types during weekly practice sessions (S1–S4). This research question evaluates whether groups receiving more specific instructions (G3 and G4) achieve consistently better debugging performance than those with abstract (G2) or no instructions (G1) at each session. Conducting multiple sessions allows us to distinguish transient improvements (e.g., one-time gains from instruction exposure) from sustained learning. Single-session evaluations risk conflating novelty effects with genuine learning, while repeated practice provides insights into whether instructional benefits persist or diminish over time. By comparing performance across weekly tests (T1–T4), we isolate the short-term effectiveness of instruction specificity from longitudinal learning trends (addressed in RQ1). We hypothesize that groups provided with detailed instructions with examples (G4) will outperform concrete but context-agnostic (G3), abstract (G2) and control (G1) groups at each weekly session, particularly in earlier stages (T1–T2), where the participants that are novice programmers may rely more heavily on explicit guidance.

2.1.3 *RQ3: What is the long-term impact of different instruction specificity levels on novice performance?*

Prior studies have focused primarily on immediate learning outcomes in programming education; however, effective instruction should produce lasting improvements in debugging ability. Thus, understanding the durability of acquired

debugging skills is crucial for long-term educational effectiveness. This research questions investigates how different instruction specificity levels affect skill retention, the three-week interval serves as a retention period to evaluate the durability of acquired debugging skills, mirroring real-world scenarios where learners may not practice skills continuously [100].

2.1.4 ***RQ$_4$: How many sessions are needed to achieve stable and optimal improvements?*** As understanding the minimum number of sessions needed to achieve stable debugging performance is essential for efficient course and curriculum design, this research question examines when learning curves plateau across different instruction specificity levels. To answer it, we analyze performance trajectories across five consecutive sessions, focusing on the rate of improvement between sessions and the point at which additional practice yields diminishing returns.

2.2 Source Code Snippets

2.2.1 Selection Criteria. To replicate real-life educational scenarios, we select code snippets from open-source projects on GitHub [35]. All code snippets used in the study were written in C# to align with participants' coursework and ensure familiarity with the programming language. Each snippet should meet the following criteria:

(1) The snippets should be self-contained and functionally independent to avoid external dependencies, they should also minimize the required domain-specific knowledge [4].
(2) The length of snippets should allow participants to comprehend and debug the code within one hour, balancing the need for sufficient complexity with participant fatigue. Since each session contains 2-3 tasks (1 practice, 1-2 tests), snippets need to be challenging enough to trigger meaningful debugging processes while ensuring participants could complete tasks without exhaustion.
(3) The complexity of snippets should include multiple method calls, logical operations, and interactions between different classes to ensure participants need to understand the code structure to locate bugs.
(4) The code must be suitable for inserting logical bugs (e.g., off-by-one error in loop termination) that maintain syntactic correctness.

2.2.2 Bug Types and Difficulty Control. Table 1 summarizes all tasks used in practice and test sessions, including snippet name, size (LOC), bug type, and a relative description of the bug's location. As designed, all bugs are logical errors positioned in mid-calculation regions, supporting our difficulty control rationale. A full task inventory with bug descriptions, code snippets detailed information, and session-assignment summaries is provided in the replication package.

We inserted a single logical bug into each code snippet. As shown in Table 1, each bug was seeded so that the program executed without crashing but produced incorrect output, requiring program comprehension for successful identification. This design choice aligns with prior work on bug taxonomies [104] and debugging task design [30, 74]: logical errors are particularly valuable for teaching because they provide no compiler error messages or locations, thereby forcing students to engage in program tracing, hypothesis formation, and systematic reasoning.

Single-bug disclosure. In this study, we operationalize *debugging* as the **bug localization phase**. Each task contained exactly one seeded logical fault located in a single statement, and participants were required to identify the **location** of the fault (function and line/statement) and provide a brief description. Code modification was not permitted, so our outcome measures reflect only the *find* phase of debugging, not diagnosis or repair. In addition, we told participants this explicitly using a standard script: "There is only one bug (in one statement) in this codebase; it is a logical bug, so

Table 1. Bug Task Inventory. All seeded bugs are logical errors positioned mid-calculation. LOC from project sources; relative locations avoid revealing exact faulty lines.

Snippet Name	LOC	Bug Type	Bug Location (relative)	Avg. Cyclomatic Complexity (AvgCC)
AStar	409	Computation / Scoring	A* score update within loop	17.25
Student-department-info	3954	Control-flow / Loop logic	DB update path (UI→DB handler)	18.06
Battleship	323	Validation / Constraint	Ship placement check (board validation)	16.00
Dijkstras	160	Control-flow / Loop logic	FindPath() main loop (mid-body)	9.00
Employee-Management-ID	484	Validation / Constraint	Mid-calculation region	6.75
Compression	430	Data structures / Frequencies	Frequency table construction	13.60
PrimerTest	473	Control-flow / Loop logic	GetPrimeFactors() method	26.25
BucketSort	157	Control-flow / Loop logic	Bucket gather/merge phase	6.00
TravellingSalesman	353	Computation / Scoring	Mid-calculation region	12.20
Prims	352	Computation / Scoring	Mid-calculation region	14.75
TanjansBridge	414	Logical error	Mid-calculation region	15.20

Full per-participant task assignment matrices, and detailed information of each code snippet (e.g., how many classes, methods, files) are provided in the replication package.

the program runs and produces output, but the output is incorrect. All you need to do is to identify where the bug is, you do not need to modify the code to fix it." When participants asked for clarification about the number or nature of bugs, the experimenter reaffirmed this statement verbatim. We acknowledge that a single-fault expectation can incentivize text-search. To mitigate this risk, we placed faults in mid-calculation regions (Table 1), varied code structure and naming across snippets, and disallowed code modification or automated search tools, thereby emphasizing program comprehension rather than keyword matching.

2.3 Pilot Study

Prior to the experiment, we conducted a pilot study with 12 participants (3 per instruction group) to validate the snippet selection and bug difficulty, assess time constraints, and refine data collection procedures. Initially starting with 15 code snippets, we discarded four that required extensive domain knowledge or spatial reasoning abilities. The pilot study confirmed that the remaining 11 snippets could be debugged within our one-hour session constraint while maintaining participant engagement. This time frame proved sufficient for completing 2-3 tasks per session without inducing excessive fatigue. The pilot study also validated our bug insertion strategy, confirming that the logical errors were neither too trivial nor too challenging to locate (correctness rate = 68% average across all pilot participants). Based on pilot participant performance and feedback, we refined our follow-up questionnaires and interview protocols to better capture debugging processes and learning progression (e.g., clarifying Likert scales to avoid misleading).

2.4 Participants

We recruited undergraduate Computer Science students at the authors' institution. From an initial pool that completed an eligibility survey, we enrolled 56 participants who met our criteria. Eligibility required completion of, or current enrollment in, at least one introductory C# course to ensure participants could navigate source code and provide informed responses.

Of the 56 enrolled, 12 participated in a pilot study; the remaining 44 were assigned to the main experiment (11 per group). The gender distribution for the main experiment was 40 male, 3 female, and 1 other. Given the small group sizes, we adopted a conservative analytic strategy: primary non-parametric tests, precision summarized via minimum detectable effects (MDEs), sensitivity checks for timing outliers, and individual-point visualizations for transparency. Full details appear in Section 2.7.

Students who did not advance to the study either did not meet eligibility requirements or did not complete scheduling. Participants received either a 1% course bonus or a $15 gift card per session. The study protocol was approved by the Institutional Review Board (IRB) at the authors' institution. Table 2 summarizes programming experience for participants in the main experiment across all groups.

Table 2. Participant Demographics by Group: Programming Experience (N=44)

		Programming Exp. (years)		C# Exp. (months)		Degree Year	
Group	**n**	**Mean (SD)**	**Min – Max**	**Mean (SD)**	**Min – Max**	**Mean (SD)**	**Min – Max**
G1	11	3.82 (2.22)	2.0 – 9.0	5.14 (10.35)	0.0 – 36.0	2.41 (1.07)	1.0 – 4.0
G2	11	4.02 (2.03)	2.0 – 9.0	3.73 (6.77)	0.0 – 24.0	2.82 (1.01)	2.0 – 5.0
G3	11	4.27 (2.82)	2.0 – 12.0	4.77 (10.38)	0.0 – 36.0	4.59 (4.78)	2.0 – 18.0
G4	11	3.95 (2.44)	2.0 – 10.0	4.77 (8.63)	0.0 – 24.0	2.68 (0.93)	2.0 – 5.0
Total	44	4.02 (2.35)	2.0 – 12.0	4.60 (8.94)	0.0 – 36.0	3.13 (2.73)	1.0 – 18.0

2.5 Instructions

Table 3 summarizes the instruction groups, their specificity levels, key features, and Step 1 examples across the different groups: G2 provides abstract goals, G3 adds structural sub-steps, and G4 incorporates code-specific elements (e.g., filenames, method calls). For tasks with different codebases (e.g., graph algorithms vs. data structures), G4 instructions were tailored to each snippet's context while retaining the same procedural framework.

(1) Group 1 (G1): Control group receiving no debugging instructions. Participants are asked to follow their own debugging procedure.
(2) Group 2 (G2): Abstract instruction group receiving general three-step debugging guidelines, with each step comprising one to two concise sentences. This abstract three-step guidelines is adapted from Koet al. [53].
(3) Group 3 (G3): Concrete, Context-Agnostic instruction group receiving specific step-by-step debugging procedures, extending G2 with detailed conceptual sub-steps. The granularity of details is adapted from Latoza et al. [57].
(4) Group 4 (G4): Our proposed concrete, context-specific instruction group receiving concrete examples with implementation details, building upon G3's instruction by incorporating specific code elements.

Full versions of the instructions given to each condition (G2 A.1, G3 A.2, and 2 examples of G4 A.3 and A.4) are included in **Appendix A A.4** (representative samples), and the complete instruction set for all snippets is available in the replication package (see Data Availability).

2.6 Study Design

2.6.1 Overall Procedure and Task Distribution. Our study comprises five sessions conducted over eight weeks, with Sessions 1-4 occurring in consecutive weeks followed by Session 5 after a three-week gap. Each session includes two types of bug localization tasks: practice tasks (Pi) and test tasks (Ti). Practice tasks provide participants with both a

Table 3. Instruction Groups: Key Features and Step 1 Examples

Group: Specificity Level	**Key Features**	**Step 1 Instruction Example**
G1: No Instruction	Participants follow their own debugging process without guidance.	N/A
G2: Abstract	General three-step guidelines adapted from Ko et al. [53].	Get a quick overview of the codebase to develop a high-level understanding of the code's architecture.
G3: Concrete, Context-Agnostic	Detailed step-by-step procedures without code-specific details. Adapted from Latoza et al. [57].	1.1. Start from the codebase's entry point. 1.2. Trace the general control flow through the codebase. • Note functions/components. • Note their locations within the code structure. • Note how they interact (e.g., method calls).
G4: Concrete, Context-Specific	Detailed steps with code-specific examples and implementation details.	1.1. Start from the codebase's entry point, which is the `Main` method in `Program.cs` on line 7. 1.2. Trace the general control flow through the codebase, for example, observe how the `graph` is initialized and populated (line 9-18), and how the `FindPath` object is created and used (line 22-25). Take stock of the codebase structure, pay attention to: • **Functions/Components**: `Graph`, `Node`, and `FindPath` classes. • **Location**: `Graph.cs` and `FindPath.cs` files. • **How they interact with each other (i.e., method calls)**: `FindPath` uses `Graph`, `CalculateShortestPath` and `GetShortestPath`.

Note: G4 instructions are tailored to each codebase but follow the same procedural framework.

Fig. 1. Experiment Design Overview.

code snippet and group-specific debugging instructions, allowing them to learn and apply debugging strategies. Test tasks, in contrast, present participants with only a code snippet, enabling us to assess their debugging performance without instructions.

From our source code snippet pool, we designate one snippet as T_0 (baseline test) that remains constant across all participants to ensure comparable initial performance measures across groups. The subsequent snippets are randomly assigned as practice tasks ($P_i, i = 1 - 5$) and test tasks ($T_i, i = 1 - 5$) for each participant. Figure 1 illustrates the study timeline, showing the sequence of tasks ($(T_0 \rightarrow P_1 \rightarrow T_1 \rightarrow ... \rightarrow P_5 \rightarrow T_5)$) and session structure over the eight-week period. Tasks are assigned per a pre-specified rotation that randomized snippet order across practice and test sessions; the complete assignment matrix (participant × session) is available in the replication package. Throughout all debugging

tasks, participants are provided with pen and paper for optional note-taking, though code modification is not permitted. This ensures consistent debugging conditions across all sessions while allowing participants to document their thought process [5, 13, 54, 58, 79].

Fig. 2. Session Procedure.

2.6.2 Session Protocol. Figure 2 outlines the structure of one experimental session. **The initial session (Session 1)** begins with an introduction to the study's background, emphasizing that participant performance is not being judged to minimize stress-induced bias. We then explain that participants will complete three bug localization tasks (one practice and two tests). Participants first complete T_0 (baseline test), followed by an optional rest period. They then proceed to P_1, where they receive their group-specific instructions alongside the code snippet. After completing P_1, we reveal the bug's location and nature, allowing participants to learn from the practice experience. Following another optional rest period, participants complete T1 with only the code snippet provided. **Subsequent sessions (Sessions 2-5)** follow a streamlined structure with two tasks: a practice task (Pi) followed by a test task (Ti), with an optional rest period between tasks. The format of each session remains consistent with Session 1's practice and test components.

2.6.3 Follow-up questions. As shown in Figure 2, after each task (both practice and test), participants complete a follow-up questionnaire asking them to provide the following:

- **Bug Location entry** (required): file name, method/function, and the specific *statement/line* believed to contain the fault.
- **Fault description** (required): a brief textual description of the logical error (what goes wrong and why).
- Task difficulty on a 5-point Likert scale (from 1 to 5 where 5 is extremely difficult).
- Debugging satisfaction on a 5-point Likert scale (from 1 to 5, where 5 means extremely satisfied).
- Stress level on a 5-point Likert scale (from 1 to 5 where 5 is extremely stressful).

For practice tasks only, participants in Group 2, 3 and 4 additionally rate:

- Instruction helpfulness on a 5-point Likert scale (from 1 to 5, where 5 means extremely helpful).
- Instruction adherence on a 5-point Likert scale (from 1 to 5 where 5 means followed very closely).

All Likert scale questions include open-ended questions for participants to explain their ratings. The complete survey is provided in the supplementary materials.

2.6.4 Performance Metrics: Correctness and Time-to-completion. Based on participants' task completion and survey responses, we measure two key performance indicators for all tasks:

- **Localization accuracy (Correctness)**: binary (1 = correct, 0 = incorrect), scored from the post-task survey. A response was scored *correct* only if (i) the location entry identified the **minimal fault-bearing statement** (i.e., the specific statement/line that contains the logical fault) in the correct file and method, and (ii) the accompanying fault description was *consistent* with that location. Entries pointing only to an enclosing construct (e.g., method/block/loop) or to a neighboring but non-faulty statement were scored *incorrect*. If the description contradicted the cited location (e.g., described a different fault), the trial was scored *incorrect*.
- Time-to-completion: Recorded without participant awareness to prevent time pressure from influencing behavior.

These metrics aim to capture the effectiveness (correctness) and efficiency (time) of participants' performance in the bug localization phase.

2.6.5 Session-End Assessment. After all tasks and task-specific surveys are completed, each session concludes with a questionnaire and interview. For Sessions 1-4, the questionnaire focuses on three key aspects:

- The utility and clarity of practice task instructions.
- Transfer of learning from practice to test tasks (specifically how participants leverage instruction information when the instruction is not provided).
- Perceived influence of practice sessions on task performance.

Session 5's questionnaire includes additional retrospective questions to assess the long-term impact of instruction on participants' debugging practices:

- Potential adaptation of provided instructions into future debugging activities.
- Observed changes in personal debugging procedures across all sessions.

Following the questionnaire, we conduct a semi-structured interview to gather detailed feedback and clarify participant responses. This data collection approach enables us to capture both quantitative performance metrics and qualitative insights into participants' debugging experiences, instruction effectiveness, and learning progression throughout the study.

2.7 Analysis Methods

For all analyses, we employ non-parametric statistical tests due to our small sample sizes ($\approx$ 10 participants per group) and non-normal data distribution, as recommended by prior works [24, 29], we use non-parametric tests as primary analyses and treat model-based estimates as robustness checks. Two key performance metrics are analyzed across all research questions: correctness and time-to-completion. For all analyses, we report individual datapoints with group summaries and 95% confidence intervals (CIs). Correctness is analyzed as a proportion (reported as %), with mean and Wald 95% CI; time-to-completion is analyzed via medians with bootstrap 95% CIs. We control multiplicity for per-session pairwise contrasts using the Holm–Bonferroni procedure[15, 33] ($\alpha = 0.05$, two-sided). To check robustness beyond our primary non-parametric tests, we fit model-based estimates with cluster-robust standard errors[18, 44] (participants as clusters): a binomial GLM for correctness and an OLS model on log-transformed time for efficiency. We conduct sensitivity analyses that flag timing outliers (MAD > 3× and IQR ±1.5× rules) and verify conclusions without excluding them, and also assess stability via leave-one-participant-out (LOPO) re-estimation for key contrasts. Finally, we summarize precision using minimum detectable effects (MDEs) to contextualize non-significant results. Descriptive

statistics and all code are included in the replication package. While our primary analysis focuses on quantitative metrics, we also incorporate qualitative data from surveys and interviews to provide context and support for our quantitative findings [23, 48].

2.7.1 ***RQ1: How do novice' debugging performance trends differ across instruction groups over multiple sessions?*** To analyze how debugging performance evolves within each instruction group, we examine within-group changes across $T_0 \rightarrow T_5$ using Friedman tests (primary), followed by Bonferroni-corrected post-hoc comparisons when applicable. In parallel with trajectories, we present per-session summaries with 95% CIs and conduct Holm–Bonferroni–adjusted pairwise contrasts between groups at each session to localize where differences emerge. As a robustness check, we fit a binomial GLM for correctness and an OLS model on log(time) with cluster-robust SEs (participants as clusters). We also report practical magnitudes (absolute percentage-point differences for correctness; minute differences for time) and verify stability of key contrasts via LOPO. To better understand the evolving performance patterns, we visualize and analyze trajectories of the two performance metrics (time and correctness) across all sessions through trend analysis and descriptive statistics.

2.7.2 ***RQ2: What is the short-term impact of different instruction specificity levels on novice performance?*** For examining the immediate impact of instruction specificity during weekly sessions ($T_1 \rightarrow ... \rightarrow T_4$), our analysis focuses on between-group differences at each test point. We employ Kruskal-Wallis tests [96] to compare performance across instruction groups at each time point, as this test is appropriate for comparing multiple independent groups without assuming normal distribution [20]. When significant differences are found ($p < 0.05$), we conduct pairwise Mann-Whitney U tests [71] to identify specific differences between groups. For the binary correctness data, we use Chi-square tests [16] to compare the distribution of correct/incorrect responses across groups [2, 70]. We also calculate effect sizes (e.g., Cohen's d) [43] for significant results to assess the magnitude of differences between groups. All pairwise p-values are adjusted via Holm–Bonferroni. We report effect magnitudes (percentage points and minutes) with 95% CIs. As a robustness check, we fit a binomial GLM (correctness) and an OLS model on log-time with cluster-robust SEs to confirm patterns observed in the non-parametric results. In addition, we also analyze survey responses on instruction helpfulness and adherence (e.g., "How helpful were the instructions?") to gather insights into why certain instruction types led to better immediate performance.

2.7.3 ***RQ3: What is the long-term impact of different instruction specificity levels on novice performance?*** Our analysis of skill retention after the three-week gap focuses on both between-group and within-group comparisons. Between-group comparisons in T_5 performance are examined using Kruskal-Wallis tests with Holm–adjusted pairwise contrasts, followed by post-hoc Mann-Whitney U tests when significant differences emerge. To analyze retention within each group, we compare T_4 and T_5 performance through both statistical tests and descriptive analysis of performance trajectories. For correctness data at T_5, Chi-square tests are employed to compare the distribution of correct/incorrect responses across groups. We corroborate descriptive conclusions with the same robustness models. Additionally, for qualitative supplements, we also analyze the feedback of retrospective questions from Session 5 (e.g., "Have you adapted any of the provided instructions into your debugging practices?") to help assess the long-term impact of instruction specificity on skill retention.

2.7.4 ***RQ4: How many sessions are needed to achieve stable and optimal improvements?*** To determine when performance stabilizes, we first analyze practical stabilization trends through visualizing the learning curves and descriptive statistics (e.g., session-wise means, variability) for all groups. This observational approach aligns with

longitudinal studies in computing education (e.g., [25, 85]) to identify the stabilization from sustained performance plateaus. We treat stability thresholds as heuristic descriptors rather than pass/fail criteria, given human-performance variability. In addition, to interpret these trends, we supplement the analysis with qualitative analysis of open-ended feedback on task difficulty and stress levels. Participants explanations helps us to gain insights about when and why stabilization occurs or falters.

We further complement this analysis with two theoretical, statistical lenses, adapted from learning curve research in industrial/organizational psychology and manufacturing [8, 32]: We measure performance variability and improvement rates within sliding windows of three consecutive sessions, as the three-session window provide sufficient data to detect patterns while maintaining sensitivity to performance trend changes [26, 52]. This approach helps us to differentiate random fluctuations from skill-development plateaus [42]. For each window, we examine two key metrics:

- Coefficient of Variation (CV): For each three-session window, CV = standard deviation/mean $\times$ 100%. We adopt a threshold of $CV < 15\%$ to indicate consistent performance, following established practices in learning curve analysis [6, 84].
- Improvement Rate: Calculated between consecutive sessions using the formula: (Performance$[Ti{+}1]$−Performance$[Ti]$)/Performance 100%. We define initial stability when improvement rates fall below 10%, aligning with the power law of practice principle in skill acquisition research [42, 77].

In addition, we employ statistical testing to validate the stability identification. We analyze the same three-session sliding windows using Friedman tests, with $p > 0.05$ indicating no significant differences in performance across the window [68]. When statistical stability is indicated, we confirm it using Wilcoxon signed-rank tests between consecutive pairs within the stable period [83].

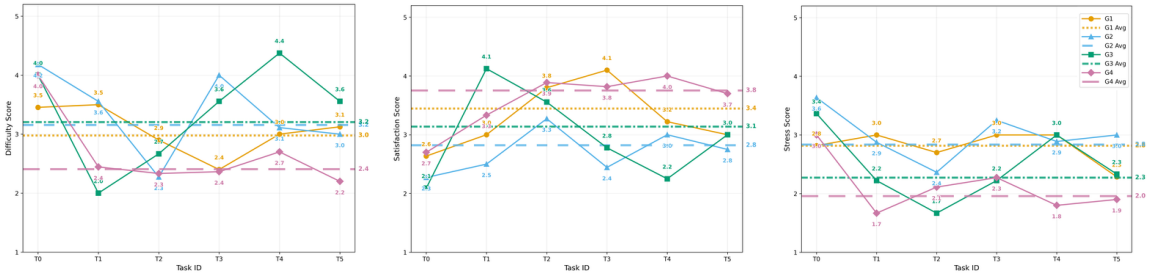
These thresholds, while not adapted and standardized in CS education yet, could help to add methodological rigor to our exploratory analysis. Since human learning involves inherent variability [77], we interpret thresholds heuristically to gain insights to stabilization patterns and validate observational findings rather than as pass/fail criteria. We apply these analyses separately to time-to-completion and correctness metrics, as these performance aspects may stabilize at different points.

3 Results

We conducted experiments with 44 participants in the experiment. Out of these participants, 41 completed all five sessions, one participant completed sessions 1-4 (missing session 5), and another two participant completed only sessions 1-2 (missing sessions 3, 4, and 5). In total, we collected data from 211 tasks ($41 \times 5 + 1 \times 4 + 2 \times 2 = 213$ bug localization tasks), measuring both correctness and time-to-completion metrics for each task.

Figure 3 illustrates participants' self-reported perceptions of task difficulty(Figure 3a), satisfaction (Figure 3b), and stress levels (Figure 3c) across all sessions, with distinct trends emerging across groups. We use distinct colors to differentiate groups: Group 1, 2, 3, and 4 are in yellow, blue, green, and pink, respectively. For each group, the solid line depicts session-specific score for each test, and the dashed line represents the group's average score across $T_1 - T_5$, excluding the baseline T_0 to facilitate cross-group comparisons.

Initially, all groups' perceived difficulty, satisfaction, and stress converge at T_0, which served as the baseline task where all participants were provided with the same code snippet. This convergence indicates that participants across groups started with similar debugging abilities (difficulty scores ranging from 3.5-4.2). After this baseline, clear differentiation emerged based on instruction type.



(a) Difficulty Scores (b) Satisfaction Scores (c) Stress Scores

Fig. 3. Test Tasks Metrics by Group. The figures show performance metrics across six tasks (T0-T5) for four participant groups. We use distinct colors to differentiate groups: Group 1, 2, 3, and 4 are in yellow, blue, green, and pink, respectively. For each group, the solid line depicts task-specific scores, and the dashed line represents the group's average score across T1-T5, excluding the baseline T0 to facilitate cross-group comparisons.

For difficulty (Figure 3a), G1 (the control group without instructions) starts at 3.5, decreases to 2.4 (T_3) as participants familiarized themselves with task formats, but rebounds to 3.1 (T_5), likely since participants forgot their self-regulated strategies. G2 (the group with abstract instructions) fluctuates moderately, reflecting challenges for novice participants to applying the abstract, general guidelines into their real debugging scenarios. G3 (the group with concrete instructions without examples) shows a sharp decline from 4.2 to 2.0 at T_1, which shows that the concrete, step-by-step instruction provides scaffolding that help participants to break the whole task down to sub-steps; however, difficulty rebounds sharply to 3.6-4.4 in later sessions, indicating the procedural steps might lose efficacy without contextual examples. G4 displays the most favorable progression, with difficulty decreasing sustained from 4.0 to 2.2 by T_5 (average: 2.4, the lowest among all groups), suggesting that contextual examples help participants overcome challenges more effectively over time.

We observed that satisfaction ratings tended to mirror perceived difficulty. At T_1, G4 reported the lowest difficulty (2.1 [95% CI 1.8–2.5]) and the highest satisfaction (3.33 [95% CI 2.57–4.09]), while G1–G3 reported higher difficulty and correspondingly lower satisfaction. This mirror pattern persisted across sessions: as G4's reported difficulty decreased and stabilized, their satisfaction steadily increased (2.7 $\rightarrow$ 4.0, $T_0 \rightarrow T_4$, with the highest average satisfaction 3.8). In contrast, G1 showed the sharpest increase in difficulty after the retention gap (T_5), accompanied by a pronounced drop in satisfaction, whereas G2 and G3 remained more stable at intermediate levels. Stress also decreases most sharply in G4 (2.9 $\rightarrow$ 1.3, $T_0 \rightarrow T_4$, also with the lowest average score 2.0), while other groups showed either persistent medium levels (G1 and G2) or only temporary relief (G3). These trends highlight a clear progression: groups receiving more instructional support (G1$\rightarrow$G4) report increasingly favorable perceptions, with G4 achieving the strongest alignment between instruction specificity and participant perceptions. The inverse relationship between perceived difficulty and satisfaction suggests that instructional scaffolding shaped not only cognitive outcomes but also affective ones. For G4, context-specific instructions simultaneously reduced task difficulty and enhanced satisfaction, consistent with theories of cognitive load and motivation. By contrast, G1–G3 participants reported more effortful experiences, which corresponded with lower satisfaction ratings. This alignment implies that well-designed instructional support can improve both efficiency and learner confidence, reinforcing the practical value of providing concrete, context-sensitive debugging guidance.

The interplay between task-specific scaffolding and longitudinal performance is further analyzed in subsequent sections for each RQ. In the rest of result section, to answer each research question, we report *groupxsession* outcomes for correctness and time with effect sizes and 95% confidence intervals, visualize individual datapoints with summary overlays, apply Holm–Bonferroni to pairwise contrasts per session, and verify robustness via outlier checks and leave-one-participant-out (LOPO) sensitivity analyses. We also summarize minimum detectable effects (MDE) to contextualize precision given the sample size.

3.1 RQ1: How do novice' debugging performance trends differ across instruction groups over multiple sessions?

In this section, we discuss how participants under different instruction groups behave during the test tasks. We compare trends for No instruction (G1), Abstract guidelines (G2), Concrete, context-agnostic (G3), and Context-specific examples (G4)across T_0 - T_5, first examine correctness trends across all sessions, followed by an analysis of time-to-completion patterns for each instruction approach. We compare *groupsession* correctness with means and 95% CIs, report Holm–Bonferroni–adjusted pairwise tests per session, and confirm robustness with cluster-robust GLMs and leave-one-participant-out (LOPO) checks. We only discuss representative results in this section to save space, the complete analysis results could be found in the replication package.

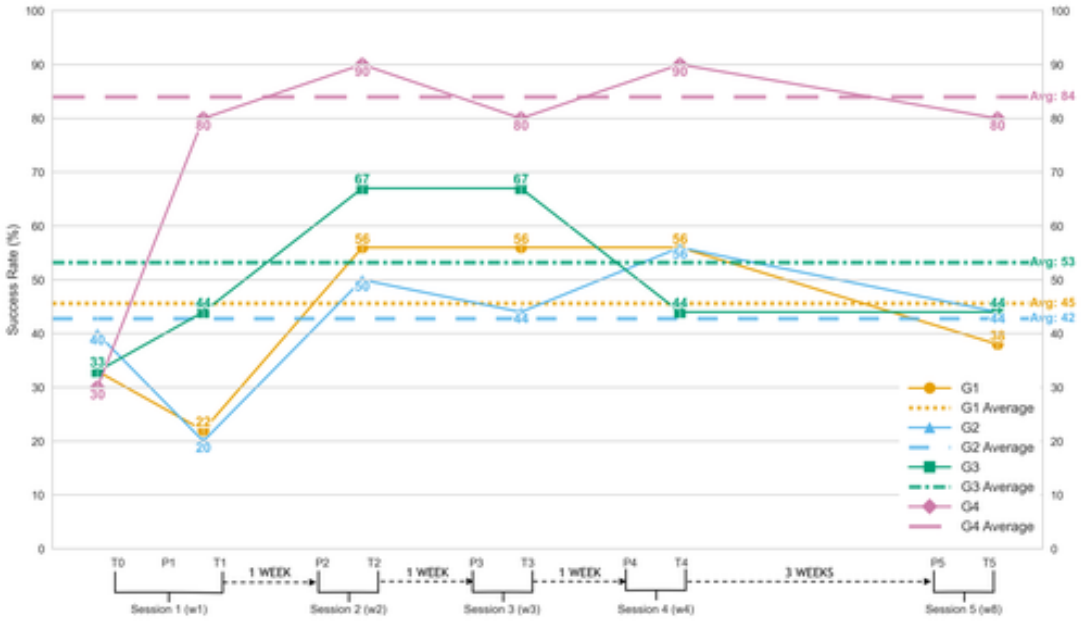


Fig. 4. Correctness Trends Across Sessions Colors: G1=yellow, G2=blue, G3=green, G4=pink.

3.1.1 Correctness. Figure 4 displays the correctness rate across all tests ($T_0 - T_5$) for all instruction groups. Consistent with our previous visualizations shown in Figure 3, we continue to use the same color-coding system to differentiate groups. Solid lines again show task-specific measurements, with dashed lines indicating group averages ($T_1 - T_5$,

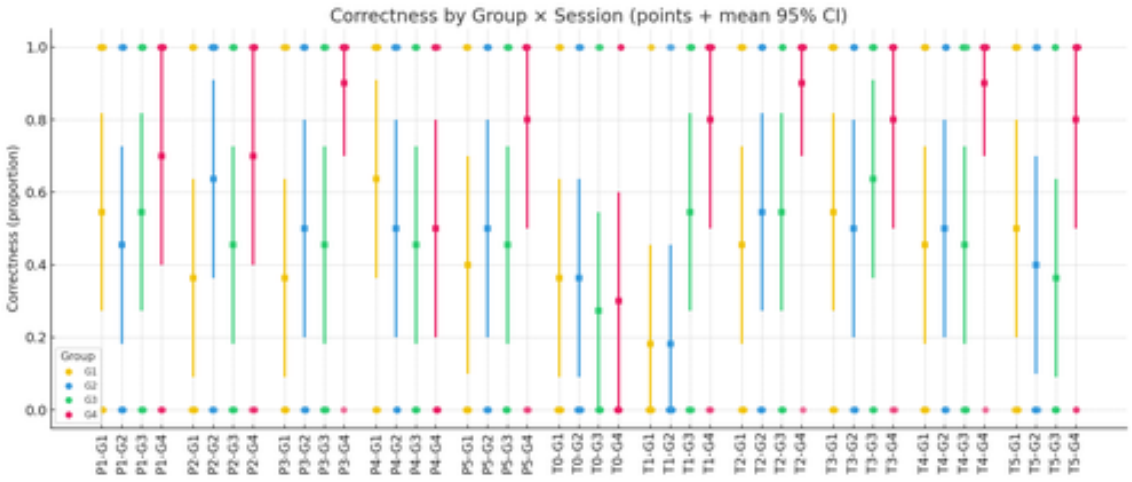


Fig. 5. Correctness by group (G1-G4) and session (T0-T5) Circles = participants; square = group mean; bars = 95% CI; Colors as in Fig. 4.

excluding baseline). Fig. 5 also displays the correctness means with 95% CIs, show individual datapoints with group summaries.

At baseline (T_0), all groups exhibit comparable performance (correctness: 30% − 40%) with no significant differences ($p = 0.973$), and wide confidence intervals that overlapped substantially (e.g., G1=36% [8%, 65%]; G4=30% [2%, 58%]). This confirms that participants in different groups are with similar initial debugging skills. This equivalence supports attributing subsequent performance differences to instructional interventions rather than pre-existing group variances."

The control group (G1) exhibits a fluctuating correctness (first dropping at T_1, then peaking at T_2, and dropping again to 38% finally) with no statistically significant changes (Friedman $\chi^2 = 4.44, p = 0.517$), with an average correctness rate at 45% (yellow dash line). This volatility reflects unguided learners' reliance on trial-and-error, described as "guessing without a clear plan" in follow-up surveys. The slight rebounds in $T_2 - T_3$ suggest tentative adaptation through trial-and-error, indicating participants might gain some strategies by repeatedly working on similar tasks. However, the drop in T_4 and low retention at T_5 reflect sporadic recall of prior attempts rather than durable skill acquisition.

G2, who received the abstract guidelines, also displays a fluctuating upward trend with limited, not significant improvements (posthoc $p = 1.0$ for T_0 vs later sessions, average: 42%). The sharp initial decline at T_1 ($d = -0.42, p = 0.786$)) aligns with high stress (3.6/5) and low satisfaction (2.5/5), as participants struggled to "translate vague instructions into actions.": Without specific guidance, these abstract instructions interfered with participants' existing debugging approaches, leading to confusion rather than improvement in bug localization performance. The fluctuated correctness after T_2 suggests that with repeated practice, participants eventually learn to interpret and apply abstract guidelines. In addition, while final correctness rate reaches 44%, between-group comparisons show no significant advantage over Group 1 ($p > 0.05$)

Group 3 shows a moderate improvement ($d = 0.22 - 0.67$) in the early sessions ($T_0 \rightarrow T_2$) but no statistical significance (Friedman $\chi^2 = 3.81, p = 0.577$), then plateaus at 56%, with a transient peak at T_3 (56%) where procedural steps aligned well with this task structure. But performance dipped again in following sessions with more complex tasks (average: 48%). This inconsistency indicates that while concrete instructions help initial learning (For example, some participants

mentioned in the interview after T_2 that "The instruction provide me a guidance about how to break the big problem into steps and sub-steps, it cut my confusion"), without context-specific examples, developers may struggle to further optimize their debugging strategies (e.g., "The steps work for simple code, but they don't help with tricky logic or the application projects, I still do not know how to trace the data flow in a larger project").

For Group 4, it achieved statistically significant improvement (Friedman $\chi^2 = 12.97, p = 0.024$) with large effect sizes ($d = 1.1 - 1.47$): Its correctness first improves sharply (↑ 50% to 80% [56%, 94%],) in T_1 then further increases to 90% at T2, demonstrating the most significant improvement with large effect sizes from T_0 to later sessions. Then G4 keeps achieving highest correctness till T_5 (average: 80% [62%, 98%]). Holm-adjusted tests confirmed these trends: at T1, G4 was significantly higher than G2 ($p\_adj = 0.005$), and at T3, G4 was significantly higher than G1 ($p\_adj = 0.008$). No other between-group differences reached significance after adjustment. A binomial GLM with cluster-robust SEs confirmed these descriptive gains: G4 contrasts were consistently positive at T1 ($\beta = 1.95, p = 0.09$), T2 ($\beta = 2.04, p = 0.08$), and T3 ($\beta = 2.20, p = 0.07$). Although some p-values were marginal, the effect directions consistently favored G4, reinforcing the descriptive findings.

This improvement is consistent with the qualitative analysis: Participants in G4 report the lowest perceived difficulty (average: 2.4) and highest satisfaction (average: 3.8), suggesting that clear, contextual examples reduce cognitive load and enhance task engagement. Although the Friedman test was not significant ($\chi^2 = 7.83, p = 0.166$), the observed gains and qualitative reports suggest that clear, contextual examples reduced cognitive load and improved engagement.

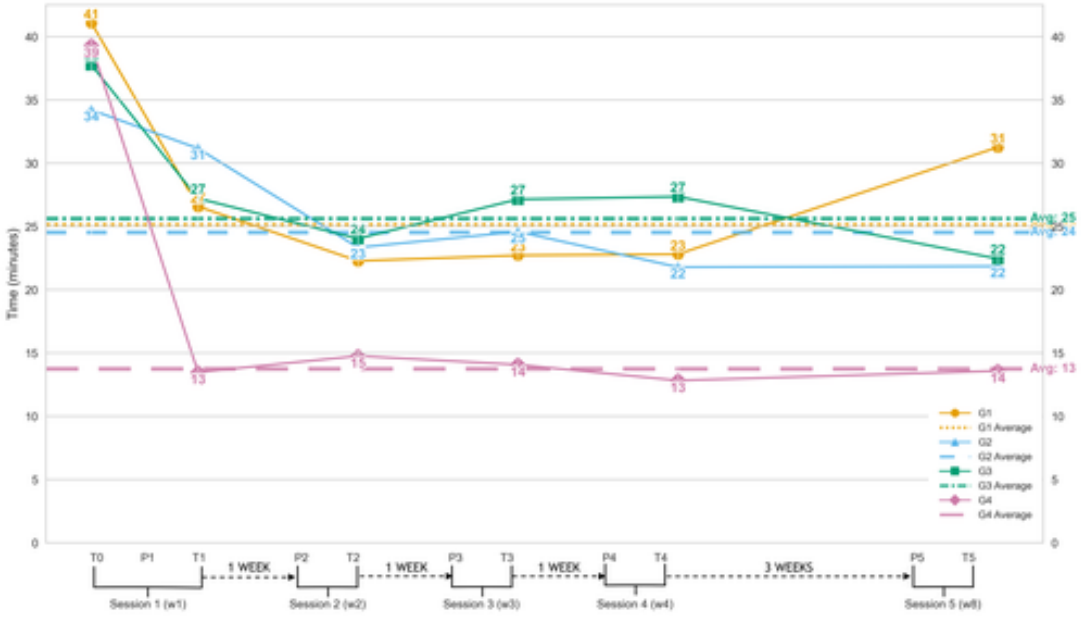


Fig. 6. Time-to-Completion Trends Across Sessions; Colors as in Fig. 4.

3.1.2 Time-to-Completion Evolution. Figure 6 illustrates time-to-completion (minutes per task) across all tests for all groups, using the same color scheme as in Figure 4. Similar as correctness, all groups exhibit comparable performance at T_0 (average completion times range from 36-41 minutes, medians: $G1 = 44[31, 55]$, $G2 = 34[21, 49]$, $G3 = 38[29, 46]$, $G4 =$

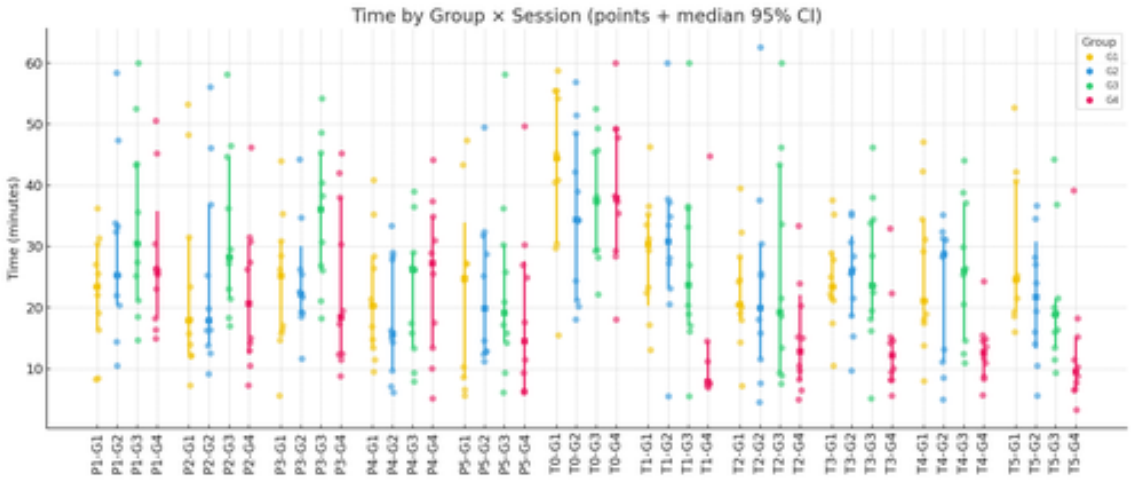


Fig. 7. Time by group (G1-G4) and session (T0-T5). Circles = participants; square = group median; bars = 95% CI; Colors as in Fig. 4.

38[29, 49]; overlapping CIs). Figure 7 shows individual data points with group summaries; squares indicate group medians and error bars show 95% bootstrap CIs.

Group 1-3 follow similar time-to-completion trends during $T_0 \rightarrow T_4$, with gradually improvements (e.g., $\approx$ 14 minutes reduction by T_2) and stabilization around 22-27 minutes at T_4, consistent with practice effects (Fig. 7). While these trends suggest experiential learning, statistical tests indicate non-significant improvements for G2 ($p = 0.255$) and G3 ($p = 0.12$). Adjusted pairwise tests among G1–G3 did not yield reliable between-group differences at the same session (Holm-adjusted). After the three-week break (T_5), G1 regresses sharply to 31 minutes ($median = 25[19, 27]$), reflecting skill decay without scaffolding, whereas G2 and G3 maintain stable performance (25 ($median = 22[14, 29]$) and 22 ($median = 19[16, 22]$) minutes, respectively), indicating that some structured guidance, even if not optimal, helps retain learned strategies. Meanwhile, G1 (control) increases to 31 minutes, suggesting that without any instructional framework, debugging strategies may deteriorate over time.

In contrast, G4 demonstrates a different trend. It first achieves the most dramatic early improvement ($39 \rightarrow 13$ minutes, $d = -2.01$, $median = 17[10, 26]$), and then maintains high efficiency (13-15 minutes) till T_5($median = 10[7, 18]$) . This rapid improvement is also statistically robust (Friedman $chi^2 = 20.93, p < 0.001$) with posthoc tests confirming significance across all sessions ($p < 0.05$). These gaps are large in practice (14-18 mins). This rapid improvement and sustained efficiency suggest that detailed, context-specific examples help developers quickly develop and retain effective debugging strategies ("The examples gave me a clear roadmap of what type of code I should focus on exactly, so I didn't waste time guessing"). In addition, Session-wise adjusted comparisons indicate a reliable G4 advantage at T_3 (G4: $median = 14[11, 21]$ vs. G3: $30[23, 37]$); Holm-adjusted $p\_adj = 0.003$). Other session-wise time contrasts do not reach adjusted significance. Participants note improved efficiency after Sessions 2 or 3 due to instruction familiarity, indicating that concrete instructions become internalized with practice.

Comparing across groups, we also conducted robustness analysis and power diagnostics. The robustness analyses supported these conclusions. LOPO stability checks showed that G4's advantage over the other groups persisted when removing any single participant, indicating the results were not driven by outliers. Similarly, excluding timing outliers

did not alter correctness outcomes. Finally, power diagnostics showed that the minimum detectable effect (MDE) for correctness was typically around 20–25 percentage points. The observed G4 advantages at T1 and T5 (35–43 points) exceeded these thresholds, suggesting the study had adequate precision to detect these meaningful differences.

RQ_1 Summary: All instruction types improve bug localization performance over time, but with different effectiveness. Context-specific instructions (G4) lead to the significantly fastest improvement and highest performance (avg 84% correctness rte, 13 minutes), while other instructions result in more gradual improvements. Qualitative data corroborates these findings: G4 reported the lowest difficulty, highest satisfaction, and minimal stress, highlighting the interplay between instructional clarity and developer outcomes. These differences remain stable across all test sessions.

3.2 RQ_2: What is the short-term impact of different instruction specificity levels on novice performance?

Our analysis of short-term instructional impacts (T_0 to T_4) reveals critical differences in debugging performance across instruction groups (No instruction (G1)–Context-specific examples (G4)), supported by both quantitative metrics and qualitative survey data.

As mentioned in the results in RQ1, all groups demonstrate comparable performance levels at T_0 (baseline) with no significant differences in either correctness rates (p = 0.973, e.g., $G1 = 36\%[8\%, 65\%], G4 = 30\%[2\%, 58\%]$) or completion time (p = 0.648, e.g., $G1 = 44[31, 55], G2 = 35[21, 48]$). After introducing different instructions, significant group differences emerge following the first practice session (T_1) in correctness ($p = 0.028$), with marginally significant time differences ($p = 0.059$). Participants receiving concrete, context-specific instruction group (G4) achieves notably higher correctness rates ($80\%[61\%, 91\%]$) compared to other groups (G1-G3, 20%-44%), with performance gaps of 36%-60%. Similarly, G4 also completed tasks twice as fast as G1 (the control group) and G3 (13 minutes vs. 27 minutes; $p = 0.015$, $d = 2.1$), and more than twice faster than the abstract instruction group (G2: 31 minutes). Pairwise tests confirmed $G4 > G2$ in correctness (Holm-adj. $p = 0.005$) and G4 faster than G1–G3 in time ($p = 0.015, d \approx 2.1$). These rapid improvements aligned with participants' experiences displayed in Figure 3: G4 reported the lowest task difficulty and highest satisfaction at T_1, suggesting that concrete, context-specific instructions enable novices to quickly grasp and apply learned debugging practices (e.g., tracing data flows or isolating logical errors) to their test case, even if they do not access the instructions during the test.

Subsequent sessions (T_2–T_4) show that G4 retains strong practical advantages, though statistical significance fluctuates across sessions. For correctness, the control group (G1) remains stagnant at 44%. In contrast, groups receiving instruction showed varied trajectories: G2 (abstract instruction) improves gradually with fluctuations (between 44% and 56%), while G3 first increases and peaking at 67% and then drop down to 44% again. Only G4 who receives the concrete instruction with contextual examples sustains highest performance (80-90%). Time efficiency reveals similar divides: The control group (G1) showed limited progress, reducing completion times modestly to 22 minutes at T_2 , then plateauing through T_4. Similarly, G2 and G3 exhibited uneven trajectories: G2 improved slightly ($31 \rightarrow 22$ by T_4), while G3 fluctuated with no consistent gains (between 24 and 27 minutes). In contrast, G4 achieved immediate and sustained efficiency. After a sharp reduction at T_1, G4 maintained consistently fast times (13–15 minutes) across all subsequent sessions. In addition, session-wise tests indicated G4 was significantly higher than G1 in correctness at T_3 (Holm-adj. $p = 0.008$) and significantly faster than G3 at T_3 (14.4 vs. 29.6 min, $p = 0.003$). At T_4, Between-group contrasts were no longer significant after adjustment, reflecting convergence or limited sample power. This pattern indicates that participants are more familiar with their debugging methodology and could follow and adapt it in different tests stably.

These trends align with participants' experiences. G4 reported steadily declining stress (1.8/5 at T_4) and high satisfaction (4.0/5), reflecting growing confidence from actionable, context-specific guidance. Conversely, G1's stagnant performance coincided with rising difficulty (3.0/5) and stress, illustrating the toll of unguided debugging. While all instructed groups (G2–G4) eventually surpassed G1 in accuracy, only G4 paired high correctness with sustained efficiency. G2's gradual gains and G3's volatility suggest abstract or mixed instructions offer limited utility for novices, who thrive instead on concrete, task-aligned examples.

RQ_2 Summary: In sum, short-term outcomes support the hypothesis: context-specific instruction (G4) yields immediate, large improvements in both correctness and time after only one session, maintains this advantage across T_1˘T_3, and remains descriptively highest at T_4. Other instruction types led to slower, smaller, or inconsistent improvements. The findings suggest concrete examples are critical for novice learners to follow and emulate.

3.3 RQ_3: What is the long-term impact of different instruction specificity levels on novice performance?

In T_5, after a three-week gap, groups receiving different instructions ((No instruction (G1)–Context-specific examples (G4))) demonstrate significantly distinct retention patterns. G1 (control: no instruction) experiences performance decline: Completion time increases from 23 minutes at T_4 to 31 minutes at T_5, while correctness drops from 44% to 38% (non-significant, p=1). This aligns with participants reporting rising difficulty (3 $\rightarrow$ 3.12) and falling satisfaction (3.22 $\rightarrow$ 3). Participants mentioned that "It's been a while so I somehow forgot my old approach and had to start from scratch again and try to recall it". These results indicate the fragility of purely self-regulated learning: without structured instruction, it is challenging for novice programmers to internalize and maintain their self-guided efficient debugging strategies over time.

Groups that received instructions without contextual examples (G2 and G3) show mixed results on both performance metrics: For G2, the correctness drops from 56% to 44% ($\downarrow$ 11%, $p = 1.0$), despite with stable completion time (22 minutes), which also revealed with G2's lowest self-satisfactions at T_5. Participants mentioned in their interview that "The guidelines is totally unhelpful since it is too general and I have to I had to figure it out on my own, I know that is a typical debugging procedure, but it has absolutely nothing to do with the actual code, it just gave me very vague instructions". Different from G2, G3 maintians strable correctness (unchanged, 44%) and reduced time significantly (27 $\rightarrow$ 22 minutes, p=0.359). This suggests procedural repetition might aid to moderate time efficiency.

For G4, it maintains its fastest completion times(14 [7,18] minutes at T_5, $p = 0.846$) and highest correctness (90% $\rightarrow$ 80%[62%, 98%], $p = 1.0$), significantly outperforming Group 1 ($p = 0.014$) and showing marginally significant advantages over Group 3 ($p = 0.053$). Comparing G4 with other groups at T_5, these gaps correspond to $\approx$ 35˘39 percentage-point advantages in correctness, and $\approx$ 8–14 minutes faster than other groups (i.e., 40–56% reductions in task time, indicating that context-specific instruction produced lasting benefits rather than decaying over time. Although the Holm-adjusted pairwise tests at T_5 did not reach adjusted significance, the descriptive differences remained large and educationally meaningful. This stability also aligns with their perceived-rankings (Figure 3): participants report the lowest task difficulty, highest satisfaction, and minimal stress.

Overall, G4's sustained performance demonstrates that context-specific instructions enable novices to automate debugging processes, retaining efficiency even after a 3-week gap. Session-end Interviews reveal that participants in G4 frequently (7/10 participants) report that the concrete, context-specific instructions become integrated into their debugging process, and they are able to customize their debugging methodology based on the type of code snippet they

are working with, leveraging experience gained from examples provided in previous practice sessions. For example, Participant 12 noted that "specific examples made it easier to remember what I should do, even weeks later."

RQ_3 Summary: Concrete, context-specific instruction (G4) demonstrates superior retention of debugging skills, maintaining both speed and accuracy after three weeks. Other instruction types (abstract and context-agnostic) supported partial retention, whereas the no-instruction control (G1) showed skill decay.

3.4 RQ_4: How many sessions are needed to achieve stable and optimal improvements?

To analyze stabilization patterns for each instruction group (No instruction (G1)–Context-specific examples (G4)), we combine identified practical trends (Figures 4 and 6), qualitative feedback, and statistical validation and discuss the results in for correctness and time-to-completion separately. Table 4 summarizes the results of our dual analytical approaches with descriptive thresholds (coefficient of variation <15%, improvement rate <10%) and non-parametric tests (Friedman/Wilcoxon), alongside observed performance levels.

Table 4. Stability Analysis Results Across Groups and Performance Metrics

Metric	Group	First Approach		Second Approach		Observed Performance
		Stability Window	CV (%) / Improvement Rate (%)	Window of No Significant Differences	p-value	Level After Stabilization
Correctness	Group 1	Not achieved	10.1 / >10	Not achieved	<0.05	38-56%
	Group 2	Not achieved	9.1 / >10	Not achieved	<0.05	44-56%
	Group 3	Not achieved	10.1 / >10	Not achieved	<0.05	44-56%
	Group 4	Not achieved*	5.4 / >10	Not achieved*	<0.05	80-90%
Time	Group 1	T2→ T3→ T4	1.0 / 0.4-2.0	T2→ T3→ T4	0.368	22-23 min
	Group 2	Not achieved	4.9 / >10	Not achieved	<0.05	22-25 min
	Group 3	Not achieved	5.7 / >10	Not achieved	<0.05	24-27 min
	Group 4	T1→ T3→ T5	3.7 / -4.7-9.6	T1→ T3→ T5	0.368	13-15 min

* Group 4 did not meet formal stability criteria but maintained consistently superior performance

3.4.1 Correctness Stability. Correctness stabilization patterns varied across groups, and when examined through our supplementary statistical tests, none of the groups met the complete formal stability criteria (CV < 15% and improvement rates < 10%). Groups without contextual guidance showed different trajectories: G1 (no instruction) performs between 38%-56% correctness with moderate variability (CV = 10.1%); G2 (abstract guidelines) reaches moderate correctness levels up to 56% but shows some variability (CV = 9.1%) reflecting challenges in applying high-level strategies. Similarly, G3 (concrete steps) maintains performance between 44-56% (CV = 10.1%) but struggles with adaptability, as participants noted, "It is difficult to adapt those steps into the tasks with larger projects". For G2 and G3, their moderate instruction adherence ratings (means ranging from 3.0-3.5) and fluctuating satisfaction levels (means between 2.5-3.5) align with these performance patterns.

In contrast, G4 demonstrates clearly superior performance, maintaining high correctness rates between 80%-90% after T_2. The temporary dip to 80% in T_3 for G4 was explained in participant feedback as related to the code base's particularly large complex method calls, followed by recovery to 90% in T_4 as they refined their approach. G4's very low variability (CV = 5.4%) suggests near-stable performance, approaching our heuristic stability thresholds. Participants in this group consistently report high satisfaction (mean > 3.8) and strong adherence to the instructions (mean > 3.7) in

later sessions, suggesting that contextual examples benefit successful internalization of debugging strategies and quick recall even after a 3-week break. One participant explained, "I start recognizing that I can trace the data flow and ignore the classes after a quick overview, I should go to the logical parts similar to what I practiced last time, it made finding bugs much easier and faster".

3.4.2 Time-to-Completion Stability. We observe distinct patterns of time performance stabilization across instruction groups. G1 achieves formal stability during window $T_2 \rightarrow T_4$, with a remarkably low CV value (1.0%) and improvement rates well within the 10% threshold (0.4-2.0%). All statistical tests within this window show no significant differences ($p > 0.05$), confirming stable performance. Unlike G1, groups G2 and G3 do not reach formal stability criteria, with CVs of 4.9% and 5.7% respectively and improvement rates exceeding our 10% threshold. These three groups converge to completion times between 22-27 minutes.

However, G4 demonstrates formal time stability from the earliest sessions, with stability window $T_1 \rightarrow T_5$, low CV value (3.7%), and improvement rates within the 10% threshold (-4.7-9.6%). Statistical tests confirm no significant differences ($p > 0.05$) throughout this window. G4's completion times remain consistently between 13-15 minutes from the earliest sessions onward, substantially outperforming other groups. Participant feedback corroborates this finding: G4 participants consistently report lower task difficulty ratings (mean < 2.5) and stress levels (mean < 2.0) after T_1, indicating that contextual examples enable participants to internalize debugging strategies and achieve superior performance stability from very early in the study.

The combined analysis reveals that time-to-completion can achieve formal and practical stability earlier than correctness, typically as early as the first session with the concrete, context-specific instruction (G4) or within 2-3 sessions for other instructions (G2-G3). In addition, the contextual examples (G4) could lead to better correctness performance (80-90%) despite not technically meeting formal correctness stability criteria. These findings suggest that curriculum design for bug localization training should prioritize contextual examples to achieve both early time efficiency and high correctness levels.

RQ_4 Summary: Time-to-completion can stabilize as early as the first session with contextual examples (G4) or within 2-3 sessions with basic instruction (G1). While formal correctness stability remained elusive for all groups, providing novices with contextual examples enables consistently superior correctness performance (80-90%) alongside formally stable and efficient completion times (13-15 minutes)

3.5 Key findings

(1) **Context-specific examples significantly enhance debugging education**: Our analysis demonstrates that concrete examples with context-specific details (G4) significantly accelerate learning outcomes, as general problem-solving strategies (G2 and G3), while providing foundational knowledge, are challenging for novices to adapt to different debugging scenarios. This suggests that debugging education can be enriched by complementing theoretical frameworks with carefully selected, realistic examples tied to specific programming contexts. Course materials and exercises could benefit from incorporating both general methodologies and targeted examples of common bug types, allowing students to build strong theoretical foundations while developing practical debugging skills. This integrated approach bridges the gap between theory and practice more effectively than either approach alone.

(2) **Even minimal practice with context-specific instruction yields significant benefits**: Our results show that even 1-2 practice sessions with context-specific examples yield significant performance improvements,

while continued practice (2-3 sessions) leads to optimal and stable performance outcomes. Thus, this flexibility allows educators to implement either brief interventions when facing curriculum constraints to still achieve meaningful learning outcomes, or comprehensive programs for maximal skill development. This aligns with cognitive load theory [22, 92, 93], where contextual examples reduce extraneous cognitive effort by scaffolding novices' attention to critical code elements. By integrating contextual examples with procedural steps, we reduce the cognitive burden of translating abstract principles into practice, enabling faster debugging mastery regardless of implementation length.

(3) **Traditional abstract debugging guidelines may be counterproductive for novice programmers**: While abstract debugging principles have long served as foundational knowledge in programming education, our findings suggest that novice developers benefit from additional scaffolding. The initial learning curve observed with abstract instruction (G2) indicates that more concrete guidance with contextual examples are needed for novices to effectively apply general guidance. Our findings suggest revising current educational approaches by maintaining abstract principles while incorporating contextual examples. Future work should investigate how such instruction impacts learners from underrepresented backgrounds, who may lack prior debugging exposure, to extend these findings toward equitable skill development.

4 Related Works

4.1 Program Comprehension and Debugging Education

Program comprehension is a fundamental skill in software engineering education, serving as the foundation for various programming activities including debugging, maintenance, and code review. Schulte et al. [88] provided a comprehensive review of program comprehension models and their applicability to novice programming education, establishing the theoretical frameworks that underpin comprehension-focused educational approaches. Building on this foundation, researchers have investigated various aspects of how novices develop comprehension skills. Lewis [61] identified five core comprehension strategies through case study analysis with one participant, revealing how novices attempt to apply expert-level comprehension techniques. The development of effective instruction methods for code comprehension has seen several important investigations. Andrzejewska and Kotoniak [7] conducted a six-month longitudinal study using eye-tracking technology, demonstrating how comprehension patterns evolve over time. Their findings showed that eye movement patterns correlate with skill development, suggesting that comprehension strategies become more sophisticated with experience.

As the field advanced, researchers began investigating more structured approaches to teaching code comprehension through high-level systematic guidenances [74] demonstrated the effectiveness of explicit debugging instruction in K12 education, finding significant improvements in both self-efficacy and debugging performance compared to control groups. Their subsequent work [73] through teacher interviews revealed a critical gap in systematic debugging instruction approaches, with most teachers relying on ad-hoc assistance methods. Building upon these findings, Bai et al. [12] evaluated a testing checklist enhanced with explicit testing strategies in undergraduate education. Their work demonstrated that structured guidance through checklists significantly improved student performance, particularly early in the learning process. This approach, which forms the basis of our G2 condition, established the value of systematic guidance while maintaining a level of abstraction that allows for student adaptation. Further advancing the field, Ko et al. [53] demonstrated that novices struggle to apply abstract debugging strategies without scaffolding, while LaTozaet al. [57] operationalized these strategies into step-by-step procedures, finding that while students value systematic

approaches, they often struggle with strategy selection. These works, which inform our G3 condition, demonstrated the potential value of more detailed, step-by-step guidance in debugging education. Recent syntheses highlight wide variability in how debugging is taught and supported for novices (Yang et al. [102]). Complementary evidence shows that structured, step-based reflection can scaffold students' debugging processes (Abu Deeb and Hickey [1]). These findings motivate our contrast between abstract guidance, context-agnostic steps, and context-specific worked examples. However, neither work address how contextual examples bridge the gap between theory and practice—a critical need identified in recent critiques of debugging pedagogy.

While existing research has made significant contributions to understanding code comprehension education, several important gaps remain. First, while different instruction approaches have been proposed and tested in isolation [57, 74], there has been limited systematic comparison of instruction specificity levels, particularly in longitudinal contexts. Second, the relationship between instruction specificity and long-term skill retention remains poorly understood. Third, the field lacks empirical evidence about how many practice sessions are needed to achieve stable performance improvements under different instruction approaches. Our work addresses these gaps by conducting a systematic comparison of instruction specificity levels across multiple sessions, examining both immediate learning gains and long-term retention. By incorporating established learning curve principles and longitudinal study methods, we provide new insights into how instruction specificity affects the development and retention of code comprehension skills.

In addition, for the scrop of tasks leveraged in the study, in this paper, we operationalize debugging as bug localization (the **find** phase), distinct from diagnosis and repair (e.g., Carver and Klahr [19]; Katz and Anderson [49]; Fitzgerald et al. [31]; Rich et al. [82]). This distinction follows recent clarifications in the CS-education literature (Kerslake [50]). The design of debugging tasks has long been debated. Early studies often seeded a single logical fault to isolate comprehension demands (Gould and Drongowski [37]; Atwood and Ramsey [10]), while contemporaneous critiques cautioned that certain bug types can encourage superficial search over deeper understanding (Brooks [17]; Sheil [91]). Follow-ups varied fault placement and type to shape deeper reasoning (Vessey [97]). We follow this tradition by using single, logical bugs in mid-calculation regions to elicit program comprehension rather than trivial detection.

4.2 Instructional Design and Worked Examples

Research on worked examples shows reliable benefits for novice learning via reduced extraneous load and clearer problem schemata [9, 94]. In instructional design, First Principles and 4C/ID emphasize combining conceptual guidance with concrete, task-embedded exemplars to support transfer [72, 95]. Our context-specific instructions (G4) instantiate these principles for debugging: they articulate the process while embedding code-level cues that guide attention to where and how to search.

4.3 Learning Curves in Software Engineering Education

Building on these instructional mechanisms, we analyze learning curves for correctness and efficiency to examine short-term gains and longer-term stabilization. The study of learning curves has provided valuable frameworks for understanding skill acquisition in technical domains. Foundational work by Anzanello [8] and Jaber [47] established core principles for analyzing learning progression, while Hutter [45] developed theoretical frameworks for understanding power-law scaling in learning curves. These works demonstrate that skill acquisition often follows predictable patterns, though the rate and stability of learning can vary significantly based on instructional approach and task complexity. Recent applications of learning curve analysis in programming education have yielded important insights. Demirtas et al. [25] demonstrated how Abstract Syntax Tree (AST) analysis can track skill development in programming tasks,

while Rivers [85] revealed that some programming concepts don't follow typical learning patterns. In the context of introductory programming, Gudmundsen [38] explored how visual programming tools affect learning trajectories when transitioning to traditional programming languages.

Methodological considerations in learning curve analysis have also evolved. Martin [67] highlighted how learning curves are sensitive to system setup changes in educational contexts, while Gallistel [32] demonstrated that group-averaged learning curves can mask individual learning patterns. These findings emphasize the importance of analyzing both group-level trends and individual performance trajectories when studying skill acquisition. Building upon these methodological insights, researchers have increasingly focused on understanding how programming skills develop and persist over time. Li [62] provided a theoretical framework for debugging education by identifying fundamental processes and novice challenges. Building on this foundation, researchers have investigated various aspects of skill development. Goletti [36] examined how tutors implement explicit instructional strategies in introductory programming, while Csendaug [89] investigated how different teaching approaches affect skill development among preservice IT teachers.

The need for understanding long-term learning outcomes has led researchers to conduct increasingly sophisticated longitudinal investigations in programming education. Andrzejewska and Kotoniak's [7] six-month study using eye-tracking technology revealed how comprehension patterns evolve naturally over time, while Lee [60] demonstrated how cognitive scaffolding affects debugging processes across different difficulty levels. These longitudinal approaches have been particularly valuable in understanding skill development patterns, with Rivers [85] showing that different programming concepts exhibit varied learning trajectories over time. Additional insights have come from extended case studies, such as Whalley's [99] investigation of novice debugging practices, which highlighted how comprehension and evidence-based activities contribute to sustained debugging success.

5 Threats to Validity

6 Threats to Validity and Generalizability

Internal validity. Each task contained exactly one logical bug and we disclosed this to participants. This expectation can incentivize superficial search. We mitigated this by (i) planting faults in mid-calculation regions, (ii) varying code structure and naming across snippets, and (iii) disallowing code modification or automated search tools, emphasizing comprehension rather than keyword matching. Residual risk remains that some participants may still have relied on heuristics. In addition, All sessions followed a fixed practice→test format, and tasks were rotated across participants. Any carryover from practice (instructions available) to test (instructions withheld) is intended—our RQs target transfer. We scheduled optional rests and bounded sessions to limit fatigue. Nevertheless, small practice effects or differential fatigue may persist. Timing was recorded unobtrusively to avoid speed pressure; we flagged timing outliers (MAD > 3×, IQR ±1.5×) and verified conclusions without excluding them. Results were unchanged in all cases.

Construct validity. We operationalize debugging as *bug localization* (find phase). Correctness required naming the *minimal fault-bearing statement* (file, method, statement) and a consistent description; repairs were not permitted. Thus, our outcomes reflect localization effectiveness/efficiency rather than diagnosis/repair quality. In addition, correctness was binary (scored per pre-registered rubric); time used medians with bootstrap 95% CIs. Likert measures (difficulty, satisfaction, stress) are self-reports and may carry response biases; we complement them with objective outcomes and triangulate with interviews.

Conclusion validity. Group sizes were small (≈10). We used non-parametric tests as primary analyses and controlled pairwise multiplicity per session via Holm–Bonferroni (α=0.05). To examine precision, we report minimum detectable effects (MDEs) for key contrasts; several observed gaps (e.g., G4 vs. others at T1/T5) exceed these MDEs, contextualizing non-significant results. Whats more, we corroborated patterns with model-based estimates using a binomial GLM (correctness) and OLS on log-time (efficiency) with cluster-robust SEs (participants as clusters). Leave-one-participant-out (LOPO) re-estimation showed that findings were not driven by any single participant. Together, these checks reduce, but do not eliminate, risks of Type I/II errors in small samples.

External validity. Participants were undergraduates at a single institution working in a lab-like setting. Caution is warranted when generalizing to other institutions, experience levels, or professional contexts. All snippets were in C# and seeded with logical (non-crashing) bugs. Results may differ with other languages, multi-fault code, or tool-augmented workflows (e.g., IDE debuggers, static analyzers). Our replication package provides full materials to enable adaptation and re-testing in other contexts. In addition, our G2-G4 materials instantiate commonly studied levels of specificity (abstract, context-agnostic steps, context-specific worked examples). Variants in tone, length, or example quality could affect outcomes; we include full texts (appendix/replication) to support replication and adaptation.

Another concerns is relevant to generalizability summary. The clearest evidence favors context-specific instruction (G4) for novice *localization* under logical-fault tasks in C#. We expect the direction of effects to hold in similar novice settings with comparable tasks; magnitude and persistence may vary with language, tooling, and curriculum design.

7 Conclusion

The results reveal that novices equipped with contextual examples achieved rapid mastery (80% correctness and 13-minute completion times after just one session) while sustaining these gains even after a three-week hiatus. In contrast, groups relying on abstract or procedural instructions exhibited slower, unstable progress, underscoring the critical role of contextual grounding in bridging theory and practice. Our findings have practical implications for computing education. First, to balance efficiency and depth in course design, educators can use brief, context-rich interventions to quickly develop debugging skills within 1-2 session and add extended practices (2–3 sessions) with contextual examples to improve accuracy. In addition, since abstract debugging guidelines alone are insufficient for novices, educators should supplement it with relevant examples to reduce cognitive load and support more equitable skill development. Based on the findings of our study, more future works of examining how contextual instruction applies to complex codebases, diverse student populations, and industry settings are needed. By emphasizing context-aware support, educators can better equip novices to approach debugging with confidence and lasting competence.

Acknowledgements

This work is partially supported by "Agency name anonymized for review". We thank all students that participated in the experiment for their time and effort.

References

[1] Lamis Abu Deeb and Thomas J. Hickey. 2021. Reflective Debugging: A Model for Teaching Debugging as a Reflective Practice. *ACM Transactions on Computing Education (TOCE)* 21, 4 (2021), 1–28. doi:10.1145/3446132

[2] Alan Agresti. 2007. *An introduction to categorical data analysis.* John Wiley & Sons.

[3] Marzieh Ahmadzadeh, Dave Elliman, and Colin Higgins. 2005. An analysis of patterns of debugging among novice computer science students. In *Proceedings of the 10th annual SIGCSE conference on Innovation and technology in computer science education.* 84–88.

[4] Joshua Akingbade, Jianhua Yang, and Mir Seyedebrahimi. 2023. Using Code Snippets to Teach Programming Languages. In *UK and Ireland Engineering Education Research Network Conference Proceedings 2023*.

[5] Amjad Altadmri and Neil CC Brown. 2015. 37 million compilations: Investigating novice programming mistakes in large-scale student data. In *Proceedings of the 46th ACM technical symposium on computer science education*. 522–527.

[6] John R Anderson. 2014. *Rules of the mind*. Psychology Press.

[7] Magdalena Andrzejewska and Paweł Kotoniak. 2020. Development of Program Comprehension Skills by Novice Programmers–Longitudinal Eye Tracking Studies. *Informatics in Education* 19, 4 (2020), 521–541.

[8] Michel Jose Anzanello and Flavio Sanson Fogliatto. 2011. Learning curve models and applications: Literature review and research directions. *International Journal of Industrial Ergonomics* 41, 5 (2011), 573–583.

[9] Robert K. Atkinson, Sharon J. Derry, Alexander Renkl, and David Wortham. 2000. Learning from Examples: Instructional Principles from the Worked Examples Research. *Review of Educational Research* 70, 2 (2000), 181–214. doi:10.3102/00346543070002181

[10] M. Atwood and H. R. Ramsey. 1978. An Experiment on the Structure of Programming Problems and Its Effect on Problem Solving. *International Journal of Man-Machine Studies* 10, 6 (1978), 609–636. doi:10.1016/S0020-7373(78)80003-8

[11] Roger Azevedo and Jennifer G Cromley. 2004. Does training on self-regulated learning facilitate students' learning with hypermedia? *Journal of educational psychology* 96, 3 (2004), 523.

[12] Gina R Bai, Sandeep Sthapit, Sarah Heckman, Thomas W Price, and Kathryn T Stolee. 2023. An Experience Report on Introducing Explicit Strategies into Testing Checklists for Advanced Beginners. In *Proceedings of the 2023 Conference on Innovation and Technology in Computer Science Education V. 1*. 194–200.

[13] Andrew Begel and Beth Simon. 2008. Novice software developers, all over again. In *Proceedings of the fourth international workshop on computing education research*. 3–14.

[14] Suresh K Bhavnani, Frederick A Peck, and Frederick Reif. 2008. Strategy-based instruction: Lessons learned in teaching the effective and efficient use of computer applications. *ACM Transactions on Computer-Human Interaction (TOCHI)* 15, 1 (2008), 1–43.

[15] Reed E Blakesley et al. 2009. Comparisons of Methods for Multiple Hypothesis Testing in Neuropsychological Research. *Neuropsychology* 23, 2 (2009), 255–264.

[16] Anne-Laure Boulesteix. 2006. Maximally selected chi-square statistics and binary splits of nominal variables. *Biometrical Journal* 48, 5 (2006), 838–848.

[17] R. Brooks. 1980. Studying Programmer Behavior Experimentally: The Problems of Proper Methodology. *Commun. ACM* 23, 4 (1980), 207–213. doi:10.1145/358841.358849

[18] A. Colin Cameron and Douglas L. Miller. 2015. A Practitioner's Guide to Cluster-Robust Inference. *Journal of Human Resources* 50, 2 (2015), 317–372.

[19] Sharon M. Carver and David Klahr. 1984. Assessing the Information Processing Skills of a Computer Program Debugger. *Cognitive Psychology* 16, 3 (1984), 346–401. doi:10.1016/0010-0285(84)90013-7

[20] Yiuman Chan and Roy P Walmsley. 1997. Learning and understanding the Kruskal-Wallis one-way analysis-of-variance-by-ranks test for differences among three or more independent groups. *Physical therapy* 77, 12 (1997), 1755–1761.

[21] Ryan Chmiel and Michael C Loui. 2004. Debugging: from novice to expert. *Acm Sigcse Bulletin* 36, 1 (2004), 17–21.

[22] Graham Cooper. 1990. Cognitive load theory as an aid for instructional design. *Australasian Journal of Educational Technology* 6, 2 (1990).

[23] John W Creswell and Vicki L Plano Clark. 2017. *Designing and conducting mixed methods research*. Sage publications.

[24] Wayne W Daniel. 1990. *Applied nonparametric statistics*. PWS-Kent Publishing Company.

[25] Mehmet Arif Demirtas, Max Fowler, and Kathryn Cunningham. 2024. Reexamining Learning Curve Analysis in Programming Education: The Value of Many Small Problems. In *Proceedings of the 17th International Conference on Educational Data Mining*. 53–67.

[26] James E Driskell, Ruth P Willis, and Carolyn Copper. 1992. Effect of overlearning on retention. *Journal of Applied Psychology* 77, 5 (1992), 615.

[27] Mithat Elçiçek. 2022. Does teaching programming have an effect on computational thinking skill? A longitudinal study. *Yükseköğretim ve Bilim Dergisi* 12, 1 (2022), 40–50.

[28] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. 2018. *How to design programs: an introduction to programming and computing*. Mit Press.

[29] Andy Field and Graham Hole. 2002. *How to design and report experiments*. Sage.

[30] Sue Fitzgerald, Gary Lewandowski, Renee McCauley, Laurie Murphy, Beth Simon, Lynda Thomas, and Carol Zander. 2008. Debugging: finding, fixing and flailing, a multi-institutional study of novice debuggers. *Computer Science Education* 18, 2 (2008), 93–116.

[31] Sue Fitzgerald, Beth Simon, Lynda Thomas, et al. 2008. Debugging: Finding, Fixing and Flailing, a Multi-Institutional Study of Novice Debugging. In *Proceedings of the 39th ACM Technical Symposium on Computer Science Education (SIGCSE)*. 163–167. doi:10.1145/1352322.1352186

[32] Charles R Gallistel, Stephen Fairhurst, and Peter Balsam. 2004. The learning curve: implications of a quantitative analysis. *Proceedings of the National Academy of Sciences* 101, 36 (2004), 13124–13131.

[33] Marco Giacalone et al. 2018. Bonferroni–Holm and permutation tests to compare health data. *BMC Medical Research Methodology* 18 (2018), 54. doi:10.1186/s12874-018-0540-8

[34] Cole Gilbert, Brian McDonald, and Michael James Scott. 2024. Exploring the Relationship between Debugging Self-Efficacy and CASE Tools for Novice Troubleshooting. In *Proceedings of the 2024 Conference on United Kingdom & Ireland Computing Education Research*. 1–7.

[35] Github. [n. d.]. GitHub. https://github.com/

[36] Olivier Goletti, Kim Mens, and Felienne Hermans. 2022. An analysis of tutors' adoption of explicit instructional strategies in an introductory programming course. In *Proceedings of the 22nd Koli Calling International Conference on Computing Education Research*. 1–12.

[37] John D. Gould and Peter Drongowski. 1974. *An Exploratory Study of Computer Program Debugging.* Technical Report RC-4887. IBM Thomas J. Watson Research Center.

[38] Dee Gudmundsen, Lisa Olivieri, and Namita Sarawagi. 2012. Reducing the learning curve in an introductory programming course. *Journal of Computing Sciences in Colleges* 27, 3 (2012), 158–159.

[39] Leo Gugerty and Gary Olson. 1986. Debugging by skilled and novice programmers. In *Proceedings of the SIGCHI conference on human factors in computing systems*. 171–174.

[40] David Hammer and Leema K Berland. 2014. Confusing claims for data: A critique of common practices for presenting qualitative research on learning. *Journal of the Learning Sciences* 23, 1 (2014), 37–46.

[41] Matthias Hauswirth and Andrea Adamoli. 2017. Metacognitive calibration when learning to program. In *Proceedings of the 17th Koli Calling International Conference on Computing Education Research*. 50–59.

[42] Andrew Heathcote, Scott Brown, and Douglas JK Mewhort. 2000. The power law repealed: The case for an exponential law of practice. *Psychonomic bulletin & review* 7, 2 (2000), 185–207.

[43] Melinda R Hess and Jeffrey D Kromrey. 2004. Robust confidence intervals for effect sizes: A comparative study of Cohen'sd and Cliff's delta under non-normality and heterogeneous variances. In *annual meeting of the American Educational Research Association*, Vol. 1. Citeseer.

[44] FL Huang and Xintong Li. 2022. Using cluster-robust standard errors when analyzing group-randomized trials with few clusters. *Behavior Research Methods* 54 (2022), 1181–1199. doi:10.3758/s13428-021-01627-0

[45] Marcus Hutter. 2021. Learning curve theory. *arXiv preprint arXiv:2102.04074* (2021).

[46] Nwannediya Ada Ibe, Rebecca Howsmon, Lauren Penney, Nathaniel Granor, Leigh Ann DeLyser, and Kevin Wang. 2018. Reflections of a diversity, equity, and inclusion working group based on data from a national CS education program. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*. 711–716.

[47] Mohamad Jaber. 2016. *Learning curves.* CRC Press Boca Raton, FL, USA:.

[48] R Burke Johnson, Anthony J Onwuegbuzie, and Lisa A Turner. 2007. Toward a definition of mixed methods research. *Journal of mixed methods research* 1, 2 (2007), 112–133.

[49] Irvin Katz and John R. Anderson. 1987. Debugging: An Analysis of Bug-Location Strategies. *Human–Computer Interaction* 3, 4 (1987), 351–399. doi:10.1207/s15327051hci0304_2

[50] E. Kerslake. 2023. Novice Debugging in Computing Education: A Review and Reconceptualization. *Computer Science Education* 33, 4 (2023), 403–428. doi:10.1080/08993408.2023.2190190

[51] Ada S Kim and Amy J Ko. 2017. A pedagogical analysis of online coding tutorials. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*. 321–326.

[52] Jong W Kim, Frank E Ritter, and Richard J Koubek. 2013. An integrated theory for improved skill acquisition and retention in the three stages of learning. *Theoretical issues in ergonomics science* 14, 1 (2013), 22–37.

[53] Amy J Ko, Thomas D LaToza, Stephen Hull, Ellen A Ko, William Kwok, Jane Quichocho, Harshitha Akkaraju, and Rishin Pandit. 2019. Teaching explicit programming strategies to adolescents. In *Proceedings of the 50th ACM technical symposium on computer science education*. 469–475.

[54] Amy J Ko and Brad A Myers. 2004. Designing the whyline: a debugging interface for asking questions about program behavior. In *Proceedings of the SIGCHI conference on Human factors in computing systems*. 151–158.

[55] Amy J Ko, Brad A Myers, and Htet Htet Aung. 2004. Six learning barriers in end-user programming systems. In *2004 IEEE Symposium on Visual Languages-Human Centric Computing*. IEEE, 199–206.

[56] Michael Lachney, Aman Yadav, Matt Drazin, Madison C Allen, and William Babbitt. 2021. Culturally responsive debugging: A method to support cultural experts' early engagement with code. *TechTrends* 65 (2021), 771–784.

[57] Thomas D LaToza, Maryam Arab, Dastyni Loksa, and Amy J Ko. 2020. Explicit programming strategies. *Empirical Software Engineering* 25, 4 (2020), 2416–2449.

[58] 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*. 492–501.

[59] Irving Lazar, Richard Darlington, Harry Murray, Jacqueline Royce, Ann Snipper, and Craig T Ramey. 1982. Lasting effects of early education: A report from the Consortium for Longitudinal Studies. *Monographs of the society for research in child development* (1982), i–151.

[60] Jiwon Lee. 2022. *Exploring the Impact of Cognitive Awareness Scaffolding for Debugging in an Introductory Computer Science Class.* Master's thesis. California Polytechnic State University.

[61] Colleen M Lewis. 2023. Examples of Unsuccessful Use of Code Comprehension Strategies: A Resource for Developing Code Comprehension Pedagogy. In *Proceedings of the 2023 ACM Conference on International Computing Education Research-Volume 1*. 15–28.

[62] Chen Li, Emily Chan, Paul Denny, Andrew Luxton-Reilly, and Ewan Tempero. 2019. Towards a framework for teaching debugging. In *Proceedings of the Twenty-First Australasian Computing Education Conference*. 79–86.

[63] Raymond Lister, Elizabeth S Adams, Sue Fitzgerald, William Fone, John Hamer, Morten Lindholm, Robert McCartney, Jan Erik Mostr¨om, Kate Sanders, Otto Sepp¨al¨a, et al. 2004. A multi-national study of reading and tracing skills in novice programmers. *ACM SIGCSE Bulletin* 36, 4 (2004),

119–150.

[64] Dastyni Loksa and Amy J Ko. 2016. The role of self-regulation in programming problem solving process and success. In *Proceedings of the 2016 ACM conference on international computing education research*. 83–91.

[65] Dastyni Loksa, Amy J Ko, Will Jernigan, Alannah Oleson, Christopher J Mendez, and Margaret M Burnett. 2016. Programming, problem solving, and self-awareness: Effects of explicit guidance. In *Proceedings of the 2016 CHI conference on human factors in computing systems*. 1449–1461.

[66] Tia C Madkins, Nicol R Howard, and Natalie Freed. 2020. Engaging equity pedagogies in computer science learning environments. *Journal of Computer Science Integration* 3, 2 (2020), 1.

[67] Brent Martin, Kenneth R Koedinger, Antonija Mitrovic, and Santosh Mathan. 2005. On using learning curves to evaluate ITS. (2005).

[68] Scott E Maxwell, Harold D Delaney, and Ken Kelley. 2004. Sensitivity analysis in longitudinal generalized linear mixed models. *Journal of Mixed Methods Research* 1, 2 (2004), 271–289.

[69] Renee McCauley, Sue Fitzgerald, Gary Lewandowski, Laurie Murphy, Beth Simon, Lynda Thomas, and Carol Zander. 2008. Debugging: a review of the literature from an educational perspective. *Computer Science Education* 18, 2 (2008), 67–92.

[70] Mary L McHugh. 2013. The chi-square test of independence. *Biochemia medica* 23, 2 (2013), 143–149.

[71] Patrick E McKnight and Julius Najab. 2010. Mann-Whitney U Test. *The Corsini encyclopedia of psychology* (2010), 1–1.

[72] M. David Merrill. 2002. *First Principles of Instruction*. Vol. 50. Educational Technology Research and Development. 43–59 pages. doi:10.1007/BF02505024

[73] Tilman Michaeli and Ralf Romeike. 2019. Current status and perspectives of debugging in the k12 classroom: A qualitative study. In *2019 ieee global engineering education conference (educon)*. IEEE, 1030–1038.

[74] Tilman Michaeli and Ralf Romeike. 2019. Improving debugging skills in the classroom: The effects of teaching a systematic debugging process. In *Proceedings of the 14th workshop in primary and secondary computing education*. 1–7.

[75] Laurie Murphy, Gary Lewandowski, Renée McCauley, Beth Simon, Lynda Thomas, and Carol Zander. 2008. Debugging: the good, the bad, and the quirky–a qualitative analysis of novices' strategies. *ACM SIGCSE Bulletin* 40, 1 (2008), 163–167.

[76] Greg L Nelson, Benjamin Xie, and Amy J Ko. 2017. Comprehension first: evaluating a novel pedagogy and tutoring system for program tracing in CS1. In *Proceedings of the 2017 ACM conference on international computing education research*. 2–11.

[77] Allen Newell and Paul S Rosenbloom. 2013. Mechanisms of skill acquisition and the law of practice. In *Cognitive skills and their acquisition*. Psychology Press, 1–55.

[78] Devon H O'Dell. 2017. The Debugging Mindset: Understanding the psychology of learning strategies leads to effective problem-solving skills. *Queue* 15, 1 (2017), 71–90.

[79] Chris Parnin and Spencer Rugaber. 2012. Programmer information needs after memory failure. In *2012 20th IEEE International Conference on Program Comprehension (ICPC)*. IEEE, 123–132.

[80] James Prather, Raymond Pettit, Kayla McMurry, Alani Peters, John Homer, and Maxine Cohen. 2018. Metacognitive difficulties faced by novice programmers in automated assessment tools. In *Proceedings of the 2018 ACM Conference on International Computing Education Research*. 41–50.

[81] Yizhou Qian and James Lehman. 2017. Students' misconceptions and other difficulties in introductory programming: A literature review. *ACM Transactions on Computing Education (TOCE)* 18, 1 (2017), 1–24.

[82] Kathryn M. Rich, C. Strickland, T. A. Binkowski, and D. Franklin. 2019. A K–8 Debugging Learning Trajectory Derived from Research Literature. *ACM Transactions on Computing Education (TOCE)* 19, 1 (2019), 1–23. doi:10.1145/3231710

[83] Toni Rietveld and Roeland van Hout. 2017. The paired t test and beyond: Recommendations for testing the central tendencies of two paired samples in research on speech, language and hearing pathology. *Journal of communication disorders* 69 (2017), 44–57.

[84] FE Ritter, LJ Schooler, et al. 2002. The learning curve. International encyclopedia of the social and behavioral sciences. In *Amsterdam: Pergamon*. 8605.

[85] Kelly Rivers, Erik Harpstead, and Kenneth R Koedinger. 2016. Learning curve analysis for programming: Which concepts do students struggle with?. In *ICER*, Vol. 16. ACM, 143–151.

[86] Rafi Santo, Leigh Ann DeLyser, June Ahn, Anthony Pellicone, Julia Aguiar, and Stephanie Wortel-London. 2019. Equity in the who, how and what of computer science education: K12 school district conceptualizations of equity in 'cs for all'initiatives. In *2019 research on equity and sustained participation in engineering, computing, and technology (RESPECT)*. IEEE, 1–8.

[87] Emmanuel Schanzer, Kathi Fisler, and Shriram Krishnamurthi. 2018. Assessing Bootstrap: Algebra students on scaffolded and unscaffolded word problems. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*. 8–13.

[88] Carsten Schulte, Tony Clear, Ahmad Taherkhani, Teresa Busjahn, and James H Paterson. 2010. An introduction to program comprehension for computer science educators. *Proceedings of the 2010 ITiCSE working group reports* (2010), 65–86.

[89] Serkan Şendağ, Ilker Yakin, and Nuray Gedik. 2023. Fostering creative thinking skills through computer programming: Explicit or integrated teaching? *Education and Information Technologies* 28, 9 (2023), 10819–10838.

[90] Kshitij Sharma, Katerina Mangaroska, Michail Giannakos, and Pierre Dillenbourg. 2018. Interlacing gaze and actions to explain the debugging process. International Society of the Learning Sciences, Inc.[ISLS].

[91] Beverly A. Sheil. 1981. The Psychological Study of Programming. *Computing Surveys* 13, 1 (1981), 101–120. doi:10.1145/356835.356841

[92] John Sweller. 1988. Cognitive load during problem solving: Effects on learning. *Cognitive science* 12, 2 (1988), 257–285.

[93] John Sweller and Paul Chandler. 1991. Evidence for cognitive load theory. *Cognition and instruction* 8, 4 (1991), 351–362.

[94] John Sweller and Mark Ward. 1989. Cognitive Load and the Worked Example Effect. *Cognitive Science* 12, 3 (1989), 257–285. doi:10.1207/s15516709cog1302_4
[95] Jeroen J. G. van Merriënboer, Paul A. Kirschner, and Liesbeth Kester. 2002. Taking the Load off a Learner's Mind: Instructional Design for Complex Learning. *Educational Psychologist* 38, 1 (2002), 5–13. doi:10.1207/S15326985EP3801_2
[96] András Vargha and Harold D Delaney. 1998. The Kruskal-Wallis test and stochastic homogeneity. *Journal of Educational and behavioral Statistics* 23, 2 (1998), 170–192.
[97] Iris Vessey. 1984. Expertise in Debugging Computer Programs: A Process Analysis. *International Journal of Man-Machine Studies* 21, 5 (1984), 459–494. doi:10.1016/S0020-7373(84)80026-9
[98] Iris Vessey. 1985. Expertise in debugging computer programs: A process analysis. *International Journal of Man-Machine Studies* 23, 5 (1985), 459–494.
[99] Jacqueline Whalley, Amber Settle, and Andrew Luxton-Reilly. 2023. A think-aloud study of novice debugging. *ACM Transactions on Computing Education* 23, 2 (2023), 1–38.
[100] Wikipedia contributors. 2024. Forgetting curve — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Forgetting_curve&oldid=1254905816 [Online; accessed 10-February-2025].
[101] Benjamin Xie, Greg L Nelson, and Amy J Ko. 2018. An explicit strategy to scaffold novice program tracing. In *Proceedings of the 49th ACM technical symposium on computer science education*. 344–349.
[102] Xu Yang et al. 2025. Debugging Education: A Systematic Literature Review. *ACM Transactions on Computing Education (TOCE)* (2025). In press.
[103] Halil Yurdugül and Petek Aşkar. 2013. Learning programming, problem solving and gender: A longitudinal study. *Procedia-Social and Behavioral Sciences* 83 (2013), 605–610.
[104] Andreas Zeller. 2009. *Why programs fail: a guide to systematic debugging*. Morgan Kaufmann.

A Instruction Texts for Experimental Conditions

This appendix provides representative samples of the instructions given to participants. **G2** presents abstract guidelines, **G3** presents concrete but context-agnostic steps, and **G4** presents two context-specific worked examples. Full variants for all tasks are available in the replication package.

A.1 G2: Abstract Guidelines

Preparation. Before examining the codebase, read its requirements to understand the intended functionality. If needed, refer back to the functionality description to avoid incorrect assumptions.

Procedure.

(1) **Get a quick overview** of the codebase to develop a high-level understanding of the architecture.
(2) **Identify and examine** sections that may contain the bug. Form a hypothesis about *where* and *what* the bug is; propose a fix and proceed to validation.
(3) **Validate your hypothesis.** If validation fails, iterate between identification and validation until the bug is localized or you exhaust candidates.

Notes. These guidelines do not reference specific files or lines; they scaffold strategic behavior only.

A.2 G3: Concrete, Context-Agnostic Steps

Preparation. Before examining the codebase, read its requirements. Refer back as needed to avoid incorrect assumptions.

Procedure.

(1) **Overview (top-down).**
  (a) Start from the program's *entry point*.
  (b) Trace the general control flow (no need for fine details initially).

(c) Take stock of: (i) functions/components; (ii) their locations; (iii) how they interact (calls, data flow).

(2) **Focused examination (narrowing).**

(a) Prioritize sections likely to contain the bug (core logic, complex calculations, loops, conditionals).

(b) For a chosen section:

(i) Trace data flow; note key variable manipulations and expected behavior.

(ii) Identify inputs and propose test inputs likely to trigger the fault (use provided inputs if unsure).

(iii) Perform a *mental trace* with those inputs; record intermediate states.

(iv) Compare observed vs. expected behavior:

- If matching: mark this section as likely correct; move to the next candidate.
- If mismatching: hypothesize the faulty statement(s); proceed to validation.

(c) If unresolved, revisit earlier candidates; then expand to less-likely sections using the same process.

(3) **Validate the hypothesis.**

(a) Assume a candidate fix in the suspected statement(s); keep other sections as-is.

(b) Redo the mental trace focusing on the fixed statement(s) (or the entire section if uncertain).

(c) If expected behavior holds, the hypothesis is supported; otherwise, revise or return to Step 2.

Notes. This procedure is concrete but code-agnostic by design; it structures how to localize faults without naming files or lines.

A.3 G4: Context-Specific Worked Example 1 (A* Path finding)

Preparation. Review the task requirements. The implementation includes components for graph handling and A* pathfinding.

Procedure.

(1) **Overview in this codebase.**

(a) Entry point: `Program.cs`, *Main* (e.g., around line 8).

(b) Observe how demos are triggered: `RunStringGraphDemo`, `RunIntGraphDemo`, `RunGridGraphDemo`.

(c) Key components and locations:

- `AStarAlgorithm` (e.g., `Calculate.cs`)
- `Graph` (e.g., `Graph.cs`)
- `IHeuristic` and heuristic implementations (e.g., `Heuristics.cs`)

(d) Interactions: `AStarAlgorithm.FindPath()` uses `IGraph` and `IHeuristic`.

(2) **Focused examination in likely fault areas.**

(a) Prioritize `AStarAlgorithm.FindPath()` (e.g., `Calculate.cs`, near line 16) because it governs pathfinding.

(b) Within `FindPath()`, trace:

- the main `while` loop (e.g., lines 28–59),
- the `gScore` / `fScore` updates (e.g., lines ~51–52),
- the handling of `openSet` and `closedSet`.

(c) Identify inputs (graph structures in `GraphFactory` of `Graph.cs`); propose cases likely to expose inconsistencies.

(d) Perform a mental trace; record intermediate states (e.g., how `gScore` and `fScore` change).

(e) Compare observed vs. expected behavior:

- If matching: check `ReconstructPath()` next.

- If mismatching: hypothesize the faulty statement(s); proceed to validation.

(3) **Validate the hypothesis.**

(a) Assume a candidate fix in the suspected statement(s) (e.g., `fScore` update); keep other code unchanged.
(b) Redo the mental trace focusing on the affected lines (or the whole method if uncertain).
(c) If expected behavior holds, the hypothesis is supported; else, revise or return to Step 2.

Notes. The steps reference concrete files/methods and plausible line ranges to scaffold attention, but no step reveals the exact faulty line.

A.4 G4: Context-Specific Worked Example 2(Student Info: Department Update)

Preparation. Review the UI and database requirements. This project uses WinForms and ADO.NET components for department updates.

Procedure.

(1) **Overview in this codebase.**

(a) Entry point: `Program.cs` (*Main*).
(b) Key UI forms/components:
- `departUpdateForm` (main update form), `Form2`, `departmentReport`, `depart_student_view`
- UI elements (buttons, text boxes, error providers)

(c) Where they live:
- `departUpdateForm.cs` (logic), `departUpdateForm.Designer.cs` (UI design)
- Other form files per project structure

(d) Interactions:
- Navigation (e.g., opening `departmentReport` from `departUpdateForm`)
- Database operations using `SqlConnection` and `SqlCommand`
- Input validation via error providers

(2) **Focused examination in likely fault areas.**

(a) Prioritize `departUpdateForm.cs` → `btnUpdate_Click` (e.g., lines 36–83), which handles the core update.
(b) Trace data flow and states:
- input validation (e.g., lines 41–60),
- database update (e.g., lines 61–83).

(c) Identify inputs: `textBoxCourse.Text` and `textBoxDuration.Text` (e.g., Course ="Computer Science", Duration ="4").
(d) Perform a mental trace following validation → update path; note variable changes and expected outcomes.
(e) Compare observed vs. expected behavior:
- If matching: move to other candidates (e.g., `btnDelete_Click`).
- If mismatching: hypothesize the faulty statement(s); proceed to validation.

(3) **Validate the hypothesis.**

(a) Assume a candidate fix in the suspected statement(s) within `btnUpdate_Click`; keep other code unchanged.
(b) Redo the mental trace focusing on the affected statement(s) (or the whole handler if uncertain).
(c) If expected behavior now holds, the hypothesis is supported; else, revise or return to Step 2.

Notes. Steps reference specific forms, handlers, and plausible line ranges to guide attention without revealing the faulty line.