Decoding Debugging Instruction: A Systematic Literature Review of Debugging Interventions

STEPHANIE YANG and MILES BAIRD, Harvard Graduate School of Education, Cambridge, MA, USA
ELEANOR O'ROURKE, Northwestern University, Evanston, IL, USA
KAREN BRENNAN and BERTRAND SCHNEIDER, Harvard Graduate School of Education, Cambridge, MA, USA

Students learning computer science frequently struggle with debugging errors in their code. These struggles can have significant downstream effects—negatively influencing how students assess their programming ability and contributing to their decision to drop out of CS courses. However, debugging instruction is often an overlooked topic, and instructors report feeling unaware of effective approaches to teach debugging. Within the literature, research on the topic is sporadic, and though there are rigorous and insightful studies to be found, there is a need to synthesize instructional approaches for debugging. In this article, we review research from 2010 to 2022 on debugging interventions. We summarize the common pedagogical approaches for learning and categorize how these target specific cognitive and non-cognitive debugging skills, such as self-efficacy and emotion regulation. We also present a summary of assessment methods and their outcomes in order to discuss intervention efficacy and directions for further research. Our sample displays a diverse variety of debugging interventions and pedagogical approaches, ranging from games to unplugged activities. An evaluation of article results also presents encouraging findings, revealing several interventions that improved debugging accuracy and learning. Still, we notice gaps in interventions addressing non-cognitive debugging skills and observe limited success in guiding students toward adopting systematic debugging strategies. The review concludes with a discussion of future directions and implications for researchers and instructors in the field.

CCS Concepts: • **Social and professional topics** → *Computer science education*;

Additional Key Words and Phrases: debugging, learning Intervention, computer science education

ACM Reference format:
Stephanie Yang, Miles Baird, Eleanor O'Rourke, Karen Brennan, and Bertrand Schneider. 2024. Decoding Debugging Instruction: A Systematic Literature Review of Debugging Interventions. *ACM Trans. Comput. Educ.* 24, 4, Article 45 (November 2024), 44 pages.
https://doi.org/10.1145/3690652

Authors' Contact Information: Stephanie Yang (corresponding author), Harvard Graduate School of Education, Cambridge, MA, USA; e-mail: szhang@g.harvard.edu; Miles Baird, Harvard Graduate School of Education, Cambridge, MA, USA; e-mail: baird.miles@gmail.com; Eleanor O'Rourke, Northwestern University, Evanston, IL, USA; e-mail: eorourke@northwestern.edu; Karen Brennan, Harvard Graduate School of Education, Cambridge, MA, USA; e-mail: karen_brennan@gse.harvard.edu; Bertrand Schneider, Harvard Graduate School of Education, Cambridge, MA, USA; e-mail: bertrand_schneider@gse.harvard.edu.

ACM 1946-6226/2024/11-ART45
https://doi.org/10.1145/3690652

1 Introduction

A major struggle for beginner programmers is wrestling with errors in their code. Although debugging—the process of locating and fixing bugs in a program—is an essential component of **computer science (CS)**, it is a highly complex and difficult task. To successfully debug a program, one must have a robust understanding of programming fundamentals [2, 53, 60], accurate mental models of the code [78, 140, 148], fault location strategies [53, 74, 88], and experience with bugs [40, 112]. Unfortunately, beginners have a fragile understanding of such concepts [116] and thus persistently struggle to locate and fix errors in their code [7, 108]. The ripple effects of these struggles are consequential—causing frustration and stress [18, 144], shaping students' assessments of their programming ability [57], and impacting their choice to remain in the field [76]. Notably, students who dropped out of CS courses cited the frustrating process of resolving even minor errors as one of their main challenges in the class [76].

Despite the importance of learning debugging skills, it is often an overlooked topic in CS education [2, 66, 100]. In a recent survey of software developers, over half stated that they never received formal education in debugging, and the majority of these reported self-learning through experience [117]. Indeed, many students learn debugging through a process of trial and error while completing their programming assignments for a course [111]. Self-learning, though, can be inefficient and places a large burden on students to seek adequate support. Further, in their analysis of debugging, Kessler and Anderson (1986) conclude that "debugging is a skill that does not immediately follow from the ability to write code," but a practice that "must be taught." [75].

Unfortunately, teaching debugging is also a difficult and complex task. While an experienced tutor is able to quickly locate and fix bugs for students, this does not necessarily translate to teaching students *how* to debug. Experts may find it difficult to teach novices since their expertise is rooted in tacit knowledge—i.e., knowledge that they are often unaware of using [132, 135]. Indeed, instructors report a limited understanding of how to teach debugging [103], ranking it as one of the most challenging concepts [83].

Taken together, debugging poses a complex pedagogical challenge, since it is both difficult to learn and also to teach. To facilitate debugging education, an essential foundation is to explore relevant research in computing education on debugging interventions. The goal of this literature review is to synthesize recent research on methods to teach debugging in the hopes of supporting instructors and researchers in the field. Through a systematic search of the literature, we identified 43 papers that conducted debugging interventions between 2010 and 2022. We first categorized these interventions by their modality and pedagogical approach to support learning. We also labeled which step in the debugging process these interventions support, and whether they address any of the non-cognitive constructs accompanying debugging. Lastly, we assessed the efficacy of these interventions and highlight opportunities for future work.

Our findings reveal a substantial diversity of intervention types, ranging from workshops to games to **augmented reality (AR)**. We also find that the majority of interventions concentrate on guiding students to identify fault symptoms and diagnose the fault, with less emphasis on teaching them to construct the problem space first and reflect on their debugging experience afterwards. Our analysis also highlights an opportunity for more focus on supporting the non-cognitive constructs related to debugging, including self-efficacy, mindset, affect, and persistence. Preliminary outcomes indicate that many interventions successfully improved debugging accuracy (correctness in fixing errors), efficiency (debugging time and steps), and self-efficacy (confidence in debugging abilities). However, these positive results did not always transfer when the intervention scaffolds were removed, suggesting opportunities for future research targeted toward learning. Additionally, interventions measuring student adoption (of debugging strategies), affect, and persistence indicated

mixed success. We contextualize these findings within the larger debugging literature and highlight opportunities for researchers and instructors.

2 Related Work

Two prior reviews provide important foundations for our current research. The first is a review of debugging conducted by McCauley et al. in 2008 [100]. Their review of research dating back to the 1970s summarizes why bugs occur, types of knowledge that aid in debugging, how novices and experts differ, and pathways forward to improve learning and teaching. In the takeaways, the authors outline a series of implications for debugging instruction, which include combating common misconceptions in coding, fostering program comprehension skills, and explicitly teaching debugging skills, such as forming and testing hypotheses. Our review builds upon and extends the work of McCauley et al. (2008), employing the PRISMA guidelines for a more systematic approach and focusing on the period from 2010 to 2022. This allows us to identify both persistent themes and emerging trends in debugging research. The current article additionally assesses intervention efficacy to highlight the most successful approaches and identify gaps in the intervention landscape. We also discuss programming tools that teach debugging, such as visualizations, games, and intelligent tutors—a category intentionally omitted in the work by McCauley et al (2008).

The second review is a framework for teaching debugging presented by Li et al. (2019) [88]. In their paper, the authors synthesize prior work to highlight the common difficulties novices face when debugging. These include difficulties understanding what the program is supposed to do, conceptually chunking the code, applying debugging strategies effectively to identify the problem, and considering alternative hypotheses of the bug location. The authors also adapt a troubleshooting framework originally proposed by Jonassen and Hung (2006) [73] to outline a general debugging process. We utilize this framework to categorize debugging interventions in our review. The authors also summarize the types of knowledge needed to successfully debug. These primary types of knowledge are: understanding the underlying programming language (domain knowledge), understanding the program itself (systems knowledge), knowledge of how to perform specific debugging tasks (procedural knowledge), knowledge of debugging strategies (strategic knowledge), and prior experience with bugs (experiential knowledge). To demonstrate the utility of this debugging knowledge framework, the authors assess several debugging tools. Their review uncovered gaps between the knowledge required for debugging and those addressed by the tools—such as teaching procedural knowledge and iteration. As an extension of this work, the current article presents a systematic review of debugging interventions, which also include non-tool formats, such as courses, unplugged activities, and learning manuals. We broaden our analysis to address important non-cognitive constructs that students commonly struggle with, such as self-efficacy and mindset. This expansion into non-cognitive factors is motivated by a growing body of research in CS education that emphasizes the importance of these constructs in learning programming [57, 77, 109]. For instance, students' mindsets and their emotional responses to errors are associated with learning [23, 109]. By including these non-cognitive factors, our review aims to provide a more holistic understanding of the debugging process and the various factors that contribute to successful debugging education. Lastly, we also evaluate intervention efficacy, highlighting important conclusions and areas for future research.

Apart from these guiding works, other literature reviews have summarized specific tools related to debugging, such as enhanced error messages and **program visualizations (PV)**. For example, Becker et al. [17] reviewed approaches to rewriting error messages to make them more helpful and readable [17]. Sorva et al. [113] reviewed PV for introductory programming education, many of which were designed to assist debugging [133]. While enhanced error messages and PV are effective tools to support or simplify debugging, they are not always designed to be tools for learning.

In the current review, we synthesize research intentionally designed to teach debugging skills—a gap noted in the literature. In 2018, Luxton-Reilley et al. conducted a large-scale review of introductory programming, with a section dedicated to debugging tools [96]. The authors surface a "notable gap" in "reviews of teaching tools [in] the areas of debugging and errors"—indicating a need to summarize research on debugging interventions.

In the current article, we present a systematic review of debugging interventions. We focus the scope of our research on papers conducted from 2010 to 2022. To adequately summarize the literature of debugging interventions, we first categorize the types of interventions that have been designed, coded by their modality and pedagogical approach for learning. We also analyze which step in the debugging process the intervention targets, by using the debugging process framework presented by Li et al. (2019) [88]. Our review uniquely summarizes approaches to support the non-cognitive aspects of debugging, including self-efficacy, affect, mindset, and persistence. Lastly, we categorize the measures that have been used to assess the efficacy of debugging interventions and their resulting outcomes. We define our core research questions as follows:

2.1 Research Questions

(1) What types of interventions have been designed to teach debugging?
(2) Which step(s) in the debugging process does the intervention target?
(3) Which non-cognitive construct(s) does the intervention target?
(4) What methods are used to evaluate the intervention?
(5) What is known about the effectiveness of debugging interventions?

3 Methods

The methods of this systematic review followed the PRISMA 2020 statement, a comprehensive guideline for documenting systematic reviews and meta-analyses. The statement details best practices for identifying, selecting, appraising, and synthesizing studies. It is widely used and focuses primarily on evaluating interventions, including those focused on education [113]. We constructed the research questions, keywords, inclusion and exclusion criteria, and data extraction processes according to these guidelines to ensure a rigorous and comprehensive approach. A PRISMA flowchart is shown in Figure 1, presenting an overview of the paper selection process.

3.1 Keywords

The scope of our literature review fell along three dimensions: debugging, education, and intervention. First, we specifically target the topic of debugging. In the preliminary rounds of our search, we experimented with different keywords to capture debugging, such as variations of "troubleshooting," "error," or "bug." However, we found that these additions generated a large number of irrelevant results and thus chose to specifically focus on permutations of the term "debug." Additionally, the review is within the domain of CS education, emphasizing the aspect of *learning* to debug. We settled on the key phrases educat*, learn*, and teach*. Lastly, we sought studies that involved interventions. This proved challenging to capture directly due to the limited use of the specific term "intervention" in studies [65]. To broaden our search in this context, we included the keywords "experiment," "effect*," and "improve" to cover papers that assessed the outcomes of interventions. Additionally, we incorporated the keyword "design" to capture studies focusing on the creation and planning of interventions. Through iterative exploration, we refined our search terms to cover the scope of the review. Our specific search phrases along the three dimensions are:

Debugging: debug*
Education: educat* OR learn* OR teach*
Intervention: intervention OR experiment OR effect* OR improve OR design

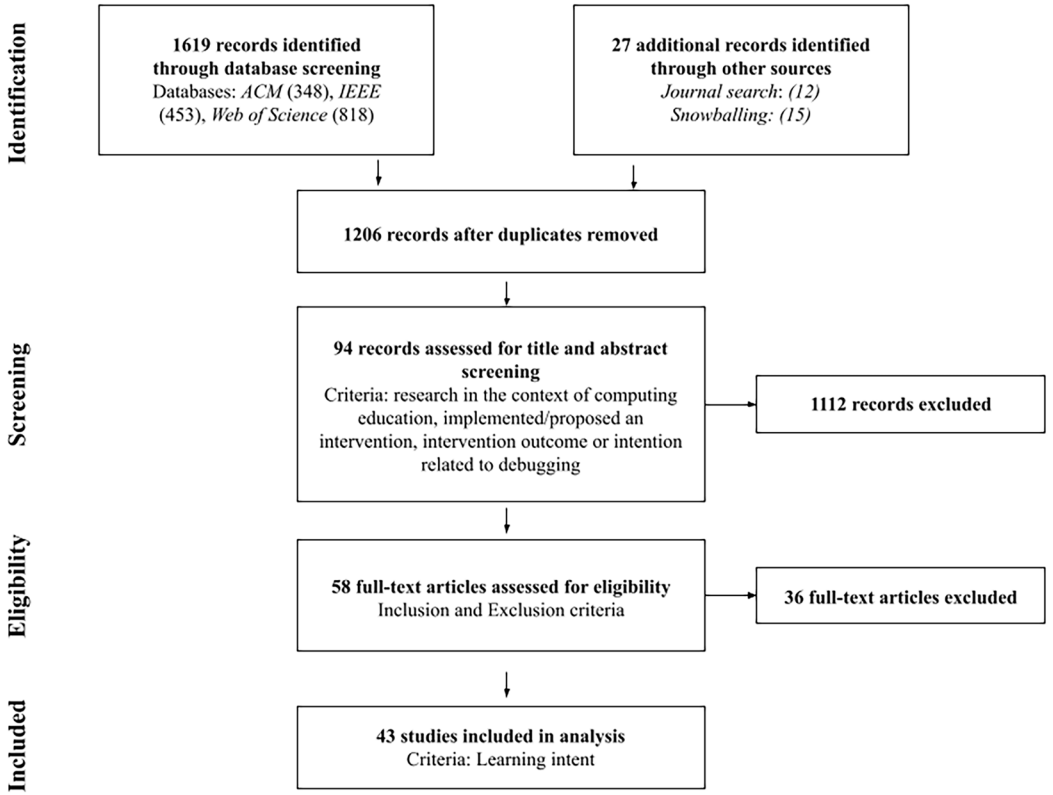

Fig. 1. *Adapted PRISMA flow diagram for paper selection.* The figure outlines the systematic literature search and screening process for the review. It depicts the number of records identified, screened, deemed eligible, and included in the analysis, with the corresponding databases and criteria at each stage. Records were identified through database screening and other sources, with duplicates removed, followed by title and abstract screening and full-text screening based on our inclusion and exclusion criteria. Full-text papers were then jointly assessed for eligibility based on the presence of a "learning intent" statement, resulting in the final included studies for analysis.

3.2 Inclusion and Exclusion Criteria

To provide an up-to-date examination of debugging interventions, our review focused on papers published between 2010 and 2022 inclusive (Table 1, Inclusion Criteria 1). Publications outside this range are not included in the formal analysis but may be included in discussion where appropriate. The scope of our article focused on interventions to teach debugging. Since our study centered on reviewing interventions, papers observing how students naturally debugged without proposing a learning support were not included in our sample. Studies were included only if they implemented or proposed an intervention (Table 1, Inclusion Criteria 2). We focused our review on papers aiming to teach students how to debug (Table 1, Inclusion Criteria 3). Thus, we did not include interventions primarily designed to scaffold debugging by making it easier, without the intent for students to learn independent debugging skills. This inclusion criteria was arguably the hardest to distinguish and evaluate. We frequently needed to carefully review the entirety of each paper to ensure accurate differentiation. Additionally, we excluded certain well-explored facets of the debugging literature, such as the enhancement of error messages, as these papers appeared to prioritize the simplification

Table 1. Inclusion and Exclusion Criteria for Screening

Inclusion Criteria	Exclusion Criteria
(1) The paper was published between January 2010 and December 2022	(1) The paper is not a research study or peer-reviewed paper (e.g., extended abstracts, posters, reviews, blogs).
(2) The research implemented or proposed an intervention	(2) The paper is not written in English
(3) The research focused on teaching debugging	(3) The paper is not accessible via university subscriptions.
(4) Either the intervention outcome or intention was related to debugging	(4) The paper is under four pages (can be 4 pages).

of debugging rather than the pedagogical aspect of instructing it. More details on enacting this inclusion criteria are provided in the database search section. Lastly, we concentrated our search on papers related to debugging specifically (Table 1, Inclusion Criteria 4). As a result, we excluded papers focused on improving programming writ-large, without mentioning a specific focus on debugging. The full inclusion and exclusion criteria is shown in Table 1.

3.3 Database Search and Paper Selection

Using our keywords as search terms, we conducted a comprehensive search on the abstracts of the ACM Full Text Collection and IEEE Explore databases, due to their coverage of computing content. Additionally, we performed a search on the Web of Science database, which covers a broader range of subject areas, such as paperss with a psychological focus. Since other relevant journals such as the *Journal of Computer Science Education* and the *Journal of Educational Computing Research* are not fully indexed in the Web of Science database, we manually searched for relevant publications to ensure comprehensive coverage.

The selection process for eligible studies involved three phases to gradually refine the pool of papers. During the identification phase, we applied the inclusion and exclusion criteria (Table 1) to the sample, focusing on the titles and abstracts of the studies. After excluding irrelevant papers, we examined the full papers and applied the inclusion and exclusion criteria. The first and second authors divided the papers equally, with each author reviewing half. Then, the authors cross-checked and reviewed each other's included papers, ensuring adherence to the predefined criteria. At this stage, we also carried out a round of snowballing for papers cited within the literature that met the inclusion criteria, following the guidelines for snowballing in systematic literature studies [145]. This was to ensure that influential and relevant debugging interventions were included in our sample.

During the screening and eligibility phase, it was often difficult to evaluate whether the paper focused on scaffolding debugging, rather than explicitly teaching debugging skills. While acknowledging the importance of these approaches and the ongoing debate surrounding scaffolding in learning, our research questions specifically focused on interventions that aimed to *teach* debugging. Thus, in the last phase, the authors jointly reviewed the papers and included those that mentioned an explicit "learning intent" statement. Through collaborative discussions, the authors identified and included papers that specifically focused on improving and teaching students' debugging skills. For example, we included the paper by Luxton-Reilly et al. (2018) because it explicitly stated its goal as "helping novice programmers improve their debugging skills" [95]. However, the article by Lee et al. (2018) was excluded during this phase since it stated that its goal was *not* "to teach debugging per se" [86]. Lastly, we encountered a few papers that described the same intervention.

We included repeated interventions if the papers measured different debugging outcomes, or if they focused on different aspects of the intervention. This occurred twice—[143, 144] and [34, 36]—with two papers describing the same intervention. Figure 1 presents a summary of our database search along with paper counts.

3.4 Paper Coding

The paper coding process involved two rounds. In the initial round, the first and second authors divided the papers for coding and extracted key information, such as publication year, subject demographic data, and programming language. In the second round, we developed a coding scheme to systematically extract data that addressed our research questions. Collaboratively, the first and second authors refined the coding scheme based on their notes from the first round of coding. They conducted **inter-rater reliability (IRR)** on a subset of the sample before proceeding to code the rest of the papers individually (see Section 3.4). The following paragraphs and Table 2 summarize our coding scheme.

RQ1. Intervention Modality and Pedagogical Approach. To address our first research question about the types of interventions, we classified studies by their modalities and pedagogical approaches. Using a bottom-up method to code for modality, we identified a set of mediums used to deliver the intervention. Some interventions were presented through in-person formats, such as through a *course* or coding workshop, *learning materials* (e.g., a debugging manual), an *unplugged activity* (e.g., embodied cognition), or *peers* (e.g., collaboration). Other instruction was delivered electronically, such as in a *coding environment*, through a *game*, or using *AR*.

To better grasp the learning mechanisms central to the intervention, we coded the pedagogical approach that these studies employed. We derived these categories from the work of Schwartz et al. (2016), who extensively classified successful pedagogical approaches based on research findings from the learning sciences [126]. While their work initially identified 26 categories, we refined our focus to those that were most prevalent within our sample and made appropriate terminology adjustments to better align with the debugging context. The primary pedagogical approaches were: *direct instruction*, where skills and strategies are explicitly taught, *deliberate practice*, which involves repeated exercises targeting a specific skill or strategy, *visualization*, often realized through flowcharts, code highlighting, or virtual overlays, *collaboration*, which expects students to co-construct and share knowledge, *embodiment*, where abstract concepts are explored and understood through learners' perceptual-motor systems, *metacognition*, which involves reflective self-regulation and self-monitoring of progress by the learner, *observation*, where strategies and techniques are modeled, often live, and *scaffolding*, where sections of an authentic activity are supported to reduce cognitive load and improve access to complex concepts. Descriptions and examples of these pedagogical approaches are discussed in the results section (see Section 4.2).

Codes for both modality and pedagogical approach were not mutually exclusive—a study could use multiple modalities and rely on several pedagogical approaches. Additionally, modality and pedagogical approach are distinct categories. For example, while an intervention may be delivered through a game format, the central mechanisms for learning may be deliberate practice of debugging strategies with scaffolded levels. When viewed together, these two coding categories offer a comprehensive overview of the diverse intervention types.

RQ2. Debugging Process Step. Our second research question focused on the specific step(s) of the debugging process that interventions targeted. To categorize the debugging process, we adapted the framework introduced by Li et al. (2019) [88], which was initially derived from a troubleshooting framework proposed by Jonassen and Hung (2006) [73]. This framework delineates four stages. During the first step, students *construct the problem space* by forming a mental model of the overall

Table 2. Coding Scheme for Analyzing Debugging Interventions

Code	Categories		Cohen's κ
RQ1. What types of interventions have been designed to teach debugging?			
Intervention modality	Course Learning material Unplugged activity Peer	Coding environment Game AR	1
Pedagogical approach	Direct instruction Deliberate practice Visualization Collaboration	Embodiment Metacognition Observation Scaffolding	0.83
RQ2. Which step(s) in the debugging process does the intervention target?			
Debugging step	Step 1: Construct the problem space Step 2: Identify fault symptoms Step 3: Diagnose the fault Step 4: Generate and verify solutions Step 5: Reflect and document		0.81
RQ3. Which non-cognitive construct(s) does the intervention target?			
Non-cognitive	Self-efficacy Mindset	Affect Persistence	0.94
RQ4. What methods are used to evaluate the intervention?			
Evaluation method	Observational Learning assessment Survey	Interview Students' written reflection None	1
Outcome	Accuracy Efficiency Adoption Perceived helpfulness Self-efficacy	Mindset Affect Persistence Other	0.76
RQ5. What is known about the effectiveness of debugging interventions?			
Effectiveness (based on significance testing)	Positive Mixed No effect	Detrimental N/A	1

The coding scheme categorizes the intervention modalities, pedagogical approaches, targeted step(s) in the debugging process, non-cognitive construct(s) addressed, evaluation methods used, and outcomes of the interventions. Cohen's kappa coefficients are provided to indicate inter-rater reliability for each category.

code structure. The next step is to *identify fault symptoms* by observing discrepancies between the intended and actual program behavior. This process may generate a series of hypotheses about the error, which aids in the third step of *diagnosing the fault*. During this stage, programmers iteratively narrow down on the exact location of the bug. The fourth step is to *generate and verify solutions*,

which involves correcting the program and testing it. Descriptions and strategies for each of these steps are further discussed in the results section (see Section 4.3).

In our coding scheme, we expanded the original framework to incorporate a fifth step: *Reflection and documentation*. This addition was motivated by the significant role of reflection and documentation in expert debugging, as highlighted by Perscheid et al. (2017) [117]. Their study revealed that 70% of software engineers regularly document their bugs, citing benefits such as code review, solving similar bugs in the future, and to teach their colleagues. Moreover, bug documentation has been recognized as a means to enhance self-awareness during the debugging process [31, 79] and to consolidate effective debugging strategies [36]. Thus, our final scheme for this code included five categories/steps.

RQ3. Non-Cognitive Constructs. Debugging is not just a series of cognitive steps, but a rich affective experience with complex metacognitive and motivational factors at play. Debugging is rightly understood to be a particularly complex emotional experience, marked by intense periods of frustration and triumph (affect) [77]. Repeatedly overcoming barriers in code progress requires persistence, with a mindset that focuses on growth rather than performance (mindset, persistence) [111]. Students also frequently self-assess their programming ability when debugging, leading to larger convictions about whether they are good or "bad at CS" (self-efficacy) [57]. Thus, we wanted to capture these constructs in our coding scheme, to code if and how papers addressed the non-cognitive factors of debugging. Taking a bottom-up coding approach, we identified four constructs represented in the wider literature of debugging and in our sample. These were: *self-efficacy*, a learner's belief in their own ability, *mindset*, a learner's outlook on failure and growth *affect*, a learner's emotional state and regulation and *persistence*, a learner's ability to persevere through difficult bugs. During the coding process, we identified whether studies addressed these in their intervention design. We coded paperss under these categories if the study mentioned how their intervention impacted these constructs, or if they tested it as an outcome. A single intervention could address multiple non-cognitive factors. Descriptions of these constructs are further detailed in our results (see Section 4.4).

RQ4. Evaluation Method and Outcome. While our first three research questions targeted the design of the intervention, our fourth research question sought to summarize how papers evaluated the intervention. To identify the evaluation approaches used to determine the effectiveness of the intervention, we coded for both the evaluation method and the debugging outcome assessed.

We first coded the evaluation method the paper used, capturing how they assessed the intervention. Gross and Powers' coding scheme is commonly employed in the field of computing education to evaluate assessments of programming environments [59]. Their categorization of assessment techniques fall under three categories: (1) anecdotal, where the authors primarily cite their personal observations, (2) analytical, which evaluates a programming environment against a specific rubric or criteria, and (3) empirical, which presents quantitative or qualitative data to evaluate an environment's impact. We did not adopt this coding scheme to categorize our papers since anecdotal and analytical assessments were seldom utilized in our sample. Instead, we modified the classifications in the empirical category to better capture assessment methods in the sample.

Our coding scheme for evaluation methods encompassed five categories: observational, learning assessment, students' written reflections, survey, and interview. The *observational* category was assigned to studies where researchers captured process data while students participated in the intervention. This label encompassed a broad category of methods, including both qualitative observations, such as researcher notes, and quantitative metrics, such as the time it took students to solve a bug. Studies that utilized a debugging test either before and after (pre-post) or only after (post) the intervention were categorized as *learning assessment*. Additionally, papers that assessed

student debugging outcomes *without* access to the intervention tools were also categorized under this label. *Survey* and *interview* codes were used when surveys or interviews were conducted to measure debugging constructs. A handful of studies also encouraged students to write reflections of their debugging experiences, which were later used to evaluate the outcomes of the intervention. These papers were categorized under *written reflection*. Since studies often conducted multiple evaluations of their intervention, these codes were not mutually exclusive.

Of equal importance to how the intervention was assessed is what was assessed. The impact of debugging interventions encompasses a range of outcomes, often associated with debugging performance, learning, or non-cognitive constructs. These outcomes were generated with a bottom-up approach, employing terminology commonly found in the debugging literature. Among these outcomes are improvements in *accuracy*, characterized by students writing fewer bugs, fixing more errors correctly, or higher scores on a debugging test. Additionally, researchers have evaluated debugging *efficiency*, such as reduced number of steps to solve a bug or shorter debugging times. Given that novices often struggle to apply debugging strategies, the extent to which taught strategies are *adopted* becomes a significant outcome of interest. In addition, students were requested to provide self-reported measures of *perceived helpfulness*, reflecting the usefulness and impact of the intervention on their learning. Lastly, researchers explored measures of *self-efficacy*, *mindset* toward errors and debugging, *affective* responses, and *persistence* (e.g., number of attempts before giving up). Collectively, this array of measures captures the nuances of debugging, providing valuable insights into the effectiveness of the interventions.

For each paper, we coded every combination of evaluation method and outcome. For example, we included separate entries if a paper assessed accuracy using both observational and learning assessment methods. The subsequent statistical results of these evaluations (see section below, Section 3.4) were also associated with the evaluation method and outcome.

RQ5. Intervention Effectiveness. Our fifth research question investigated the effectiveness of debugging interventions. Several papers within our sample conducted qualitative studies, employing design-based research methods to highlight important outcomes. Additionally, many papers did not conduct controlled experiments or provide effect size data. As such, summarizing effectiveness with a meta-analysis was not reasonable to capture the rich findings in our sample. Instead, we employed a combination of qualitative and quantitative methods to summarize the efficacy of these interventions.

For papers that did not experimentally assess their intervention, we summarized the results during the coding process, which we report qualitatively in our results section. For the papers that conducted empirical testing and reported statistical significance, we utilized coding categories derived from D'Angelo and Schneider (2021) to capture the result. We categorized the intervention as having a positive, mixed, detrimental, or no effect on the debugging outcome, based on whether the hypothesis testing was significant ($p < 0.05$) or not. Specifically, we coded the results of the significance testing for each combination of evaluation method and outcome examined in the paper. Thus, each intervention may receive multiple effectiveness codes. Significance testing could be conducted with experimental, quasi-experimental, or non-experimental study designs. If the experimental group performed significantly better than the control group on an outcome, or if students performed significantly better after the intervention or with the tool, this was coded as a *positive effect*. If the opposite was true, the intervention was coded as having a *detrimental effect*. In cases where the testing was non-significant, the intervention was coded as having *no effect* on the outcome. If a paper assessed an outcome in multiple ways (e.g., employing various observational accuracy calculations) and obtained a combination of positive, detrimental, or no-effect results, we coded this as having a *mixed effect*.

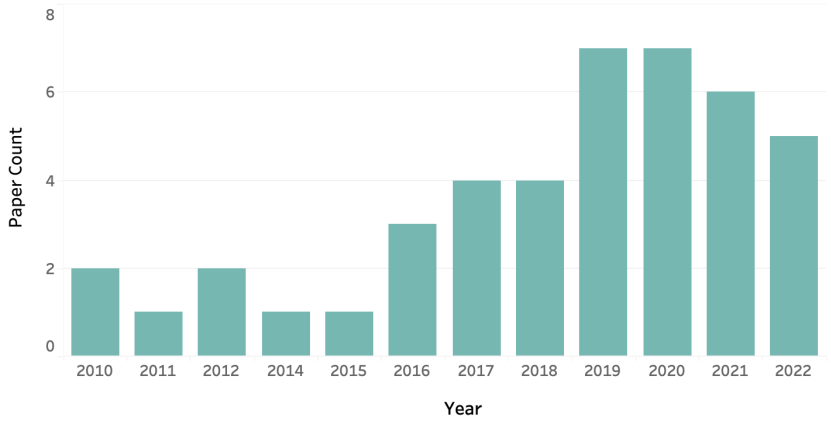

Fig. 2. *Paper counts by year.* The figure shows how many papers in our sample were published each year. There were no papers published in 2013.

Reliability. After establishing an initial coding scheme, the first and second authors jointly coded five papers, representing 12% of the sample. They iteratively discussed and refined the codes during this collaborative phase. Once they achieved a satisfactory level of agreement, they coded an additional seven papers (16% of sample) together, to assess IRR. The average IRR was $\kappa = 0.91$, indicating a high-level of agreement [101]. Detailed reports on the individual IRRs for each code are shown in Table 2, with all codes demonstrating strong to almost perfect agreement. After achieving an acceptable consensus, the first two authors divided the remaining papers to code individually. If any uncertainties arose in the individual coding process, the first and second authors discussed these collaboratively to reach a consensus.

4 Results

We examined a total of 43 papers that were published across 13 years from 2010 to 2022 (Figure 2). Appendix A (Table A1) presents the author, year, subject education level, and programming language for each paper in our sample. There appears to be a trend of growing interest in debugging interventions as an area of research. For the first 9 years in our sample window, an average of 2.3 papers were published annually except 2013 from which no papers were included. In 2019, there is a notable change with seven papers published, and the remaining 3 years averaging six papers per year.

Eight education-levels are present in the studies, the most common being university (58%, 25 papers), followed by high school (19%, 8 papers), middle school (16%, 7 papers), elementary (12%, 5 papers), adults (7%, 3 papers), and an unspecified level (2%, 1 paper). Of note is an increasing focus on interventions for K-12 and adult learners. In the 2010–2018 period, 22% of the studies target non-university learners in their interventions, while from 2019 to 2022, 56% of studies target non-university learners.

The three most common languages in the interventions are Java (28%, 12 papers), block-based (23%, 10 papers), and Python (16%, 7 papers). C++ and C account for 21% of the papers analyzed (9 papers), while Javascript is used in 1 paper. One of the two tangible languages present in our sample is Robo-Blocks, a system targeting young learners through which a robot's motion is controlled by assembling physical command blocks [130]. In addition, there are nine instances of virtual block-based languages in our sample. Block-based languages like Scratch are first targeted in 2017 and remain a consistent part of the literature from 2019 onward. Conversely, the share of papers targeting Java declines over time, and none of the papers in the last 2 years of our window use it.

Table 3. Intervention Modalities

Modality	Papers	Count
Course/Workshop	[4, 24, 34, 36, 42, 47, 52, 80, 104, 119, 131, 141, 143, 144, 146, 149]	16
Manual/Learning material	[22, 47, 54, 143, 144]	5
Unplugged activity	[4, 52, 130]	3
Peer	[52, 107, 149]	3
Coding environment	[1, 3, 10, 28, 33, 46, 51, 58, 67, 71, 85, 95, 97, 105, 118, 123, 136, 146]	18
Game	[22, 35, 71, 85, 91, 105]	6
AR	[8, 32, 121]	3

Categorization of debugging interventions by the delivery modality.

4.1 Modalities

To summarize the spectrum of debugging interventions in our sample, we began by classifying the interventions based on the modalities they employed. In our coding scheme, we classified intervention modality according to its delivery method, which included courses, learning materials, unplugged activities, peer interactions, coding environments, games, and AR (Table 3).

Many papers in our sample rely on instructor-led coursework to deliver an intervention, and 37% of our sample (16 papers) include this modality, coded as "Course/Workshop." One such example is Ko et al. (2019) who studied the effect of a 5-week workshop for high school students [103]. Students were taught a "debug" strategy by brainstorming hypotheses about the bug location, and systematically investigating each one. Instructors used metaphor, modeling, and a strategy tracker to teach this approach. Another modality was manuals or learning materials, which accounted for 12% of our sample (5 papers). In our coding, learning materials were generally static resources used to guide learners, such as posters and manuals. Garcia et al. created a debugging manual presenting both cognitive strategies to guide the debugging processes and non-cognitive strategies to help learners manage their affective responses [54].

Unplugged activities, where students use physical objects or interactions rather than computers to explore computational thinking and debugging, accounted for 7% of our sample (3 papers). For example, to help students better understand the code execution, Ahn et al. asked elementary students to act out the code through embodiment techniques. A handful of papers explored the effect of collaboration (3 papers, 7%) and peer interaction as the mode of learning. For example, Zhong and Li compared the performance of paired programmers with that of individual programmers in a summer camp class [149].

Many papers used digital modalities for their interventions. Coding environments, the most common example, accounted for 42% of our sample (18 papers). We considered tools embedded within IDE to fall under this category, along with tutorial programs having simple coding interfaces. This included work such as Carter (2015) where an intelligent tutoring system dynamically broke code to generate debugging exercises for students [28]. Another digital modality, games, applied to 14% of our sample (6 papers). This category included interventions using bespoke computer games to teach debugging, such as Gidget (Figure 3), where learners guide a personable robot through debugging puzzles [85]. AR, using digital devices to embed information and interactions in the physical world, accounted for 7% of our sample (3 papers). Chung and Hsiao presented

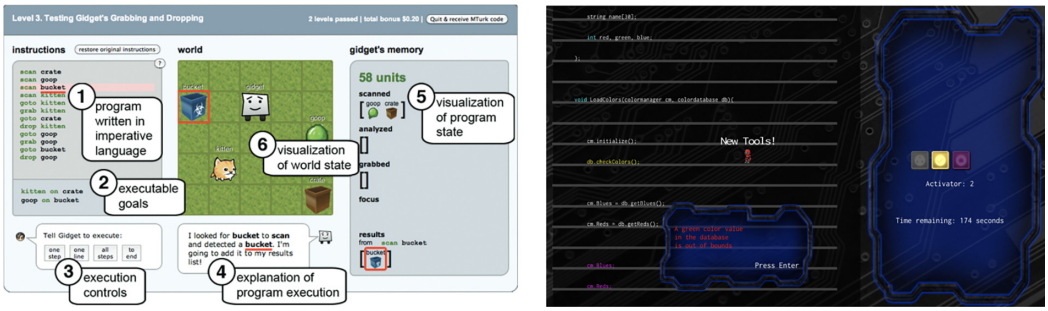

Fig. 3. *Examples of debugging games.* The example on the left (screenshot from [85]) depicts the debugging game, Gidget. Learners solve a series of debugging puzzles and interact with a personified compiler. The example on the right (screenshot from [105]) depicts the debugging game, Robobug. The game introduces a series of debugging strategies that learners must use to progress the narrative.

Table 4. Pedagogical Approaches

Pedagogical Approaches	Papers	Count
Direct instruction	[24, 36, 42, 47, 54, 80, 104, 105, 131]	9
Deliberate practice	[10, 28, 30, 42, 47, 51, 67, 71, 85, 91, 95, 104, 105, 141, 146]	15
Visualization	[8, 10, 32, 35, 46, 67, 97, 104, 121, 123, 136]	13
Collaboration	[22, 32, 52, 107, 130, 141, 149]	7
Embodiment	[4, 121, 130]	3
Metacognition	[1, 4, 24, 30, 34, 36, 42, 52, 80, 91, 95, 107, 130, 141, 143, 144]	16
Observation	[22, 24, 30, 36, 80, 104, 119, 143, 144]	9
Scaffolding	[3, 4, 10, 22, 28, 32, 42, 46, 47, 51, 52, 54, 58, 67, 71, 80, 85, 91, 95, 105, 118, 119, 121, 130, 131, 136, 143, 144]	28

Categorization of debugging interventions by the pedagogical approach.

students with a three-dimensional debugging task and allowed some participants to inspect the task environment from different angles using AR [32].

4.2 Pedagogical Approaches

Debugging instruction is often difficult because it must scaffold the experiential nature of learning a skill [132]. To answer the first research question and understand what types of interventions have been designed to teach debugging, we summarized the primary pedagogical approaches used in the included studies, which are couched in methods drawn from the learning sciences (Table 4). We describe the learning mechanisms behind these pedagogical approaches below and discuss how they were utilized in our sample.

Direct Instruction. Based on the observation that few CS courses explicitly teach debugging [66], a common intervention is to provide systematic instruction on the debugging process and debugging strategies. Decades of research dating back to the 1970s has focused on how experts approach bugs [100]. They tend to follow a systematic process akin to the scientific method, beginning with forming hypotheses and iteratively testing them to pinpoint the error location [140]. To

support students in developing these skills, several interventions have attempted to explicitly teach this process. In an early study, Carver and Risinger (1987) showed that explicit instruction on a systematic debugging process resulted in significant improvement in debugging ability [29].

In our sample, nine papers (21%) utilized this approach. For example, Michaeli and Romeike (2019) conducted a study that replicated the findings by Carver and Risinger (1987). They presented a systematic debugging process to students on a poster, walked through each step and used it to work through an example. Students subsequently practiced applying this approach through targeted debugging exercises. Similarly, Böttcher et al. employed a "just-in-time telling" instructional method [24]. Students were first tasked to read a text on debugging, which was followed by a live-coding lecture modeling an explicit debugging approach. In addition to instructor-led formats, interventions that supplied textual guidance for debugging procedures or strategies were also coded under direct instruction. For example, Garcia et al. created a debug manual with specific explanations and procedures for executing common debugging strategies [54]. Their manual also included direct instruction on how to regulate one's emotions while debugging, citing theories of emotional awareness.

Deliberate Practice. Direct instruction is frequently accompanied with opportunities for students to practice the strategies and procedures that they learned. To be most effective, deliberate practice should be goal-oriented, repeated, and situated in a rich feedback loop [48]. This practice typically occurs outside the context of the authentic activity, with students working on specific debugging problems or skills without actively coding [126]. In an early study, Chmiel and Loui (2004) showed that students who completed more debugging exercises were subsequently more efficient at debugging their own code [31].

In our sample, 15 papers (35%) utilized some form of deliberate practice. Beyond familiarizing students with common errors, these exercises aimed to cultivate *strategic* and *procedural* knowledge. Robobug (Figure 3), a debugging game, taught students how to use specific debugging strategies, such as print statements, decomposition, and code tracing [105] and provided exercises to practices these strategies. Other exercises can teach the procedural knowledge of how to perform sub-tasks, such as using the debugger, creating test cases, or inspecting memory. In an exercise called Binary Bomb, students practice setting breakpoints in strategic locations of assembly code to defuse a hidden bomb in the program [26, 51]. When crafting the Ladebug programming environment, the authors' primary aim was to establish a controlled and simplified setting for students to practice using the debugger while working through various pre-designed exercises [95]. Their intent was for students to transfer these procedural skills to debugging code in their own **Integrated Development Environments (IDEs)**. Other forms of deliberate practice in this category took place in a game settings [71, 85, 91, 105] or as exercises in class [42, 47, 104, 141, 146].

To provide the opportunity for repeated and scaffolded practice, several interventions generated banks of buggy code for students to practice on. Most frequently, these exercises were created by instructors or researchers who were familiar with stereotypical bug patterns [10, 30, 51, 67, 95, 146]. However, Fields et al. explored asking *students* to generate buggy code for each other to solve. Apart from affording students opportunities to debug code, the authors theorized that as students generated flawed code, they would gain a greater sense of agency over their learning and the act of making mistakes [52]. Two papers in our sample also explored generating buggy code dynamically, either through pre-configured bug patterns [28] or in response to errors which learners encountered in previous exercises [71]. Among papers focused on deliberate practice, exercises primarily tasked students to debug pre-existing faulty code. While this provides targeted debugging practice, it also raises an important consideration, since debugging someone else's code can qualitatively differ from debugging one's own code [144].

Visualization. PV have long-been a common pedagogical tool to help students learn CS [133]. Visualizing the program execution and control flow can be particularly helpful for novices, who often struggle to chunk relevant information and build a mental-model of the code [89, 140]. Encouraging students to generate their own visualizations is another effective learning mechanism, since they must map out the intricacies of the underlying code logic [68, 69]. Early work in PV for novices can be traced back to the 1980s, and contemporary software solutions such as Jeliot [106], BlueJ [81], and Python Tutor [61] have become widely adopted and actively maintained. In 2013, Sorva et al. conducted a comprehensive survey of pedagogical programming visualization tools [133]. Their analysis revealed that these visualizations predominantly contribute to learning. However, it also highlighted that the research prototypes often have short lifespans and lack thorough investigation. Notably, there remains a gap in the discussion of visualizations tailored to support debugging, which is often recognized as a separate skill from programming itself.

Out of the papers we sampled, 13 (30%) made use of visualizations as tools to support debugging. Four designs adhered to traditional conceptions of PV, which typically provide snapshots of the code's execution and allow users to step through and set breakpoints [33, 35, 67, 123]. For example, Santos (2018) created PandionJ, a pedagogical debugger designed to resemble teacher-drawn diagrams of student code [123]. When the code loops through an array, the canvas displays the array values and indices, along with arrows and bars indicating the current step of the loop and the loop bounds. These additional visual indicators allow students to understand the current and future states of the variables when debugging their loops. Another category of visualizations provided visual cues to steer learners to the fault location [10, 46, 136]. Ardimento et al. (2019) directed students' investigations by visualizing possible fault paths in a program [10], while Edminson and Edwards (2020) used **statistical fault localization (SFL)** to highlight lines of suspect code when errors occurred [46]. Three designs also used AR to visualize hidden code states and enable collaboration, particularly for physical computing environments [8, 32, 121]. For example, Alrashidi et al. (2017) created an AR application for a robot programming task [8]. When students encountered an error, they could point their tablet at the robot to view an overlay of the robot's actions, behaviors and sensor values in real-time. This external representation of hidden code states allowed them to compare expected and actual outputs of the code.

Collaboration. During collaborative learning, students work together to construct knowledge and solve problems [134]. Pair programming is shown to be effective in boosting course retention, quality of code, and learning gains [63]. When working in pairs, the cognitive load is shared between partners, allowing them to exchange and build on each other's ideas [27]. While collaboration has been extensively studied in programming, less research has focused on specific subskills of the practice [63], including debugging.

In our sample, seven papers (16%) incorporated aspects of collaboration in their intervention, primarily through pair debugging. Murphy et al. analyzed the transactive discourse between collaborators as they worked through debugging challenges together [107]. They hypothesized that collaboration could be especially effective in the context of debugging, since both parties could share the cognitive load and search for errors. Additionally, "thinking aloud" with their partners could help to thoughtfully reason through the problem, while also providing opportunities to vent and process negative emotions associated with the debugging process. Fields et al. (2021) and DeLiema et al. (2019) also noted the potential regulatory benefits of collaboration and asked students to share moments of struggle while debugging with their classmates [36, 52]. Collaboration was often employed within physical computing contexts or using AR [32, 52, 130]. While computer-based programming paradigms often require collaborators to alternate roles between driving and

navigating, physical computing offers both parties the opportunity to simultaneously discuss and manipulate the code.

Embodiment. Embodied cognition harnesses the intelligence stored in the perceptual-motor system to help make sense of abstract concepts [15]. When programming activities incorporate bodily engagement, learners can enhance their code comprehension [72] by embodying the code execution. Externalizing the program execution beyond mental visualization also alleviates cognitive burden, allowing students to better focus on the programming task at hand [82]. Since debugging requires complex abstract thinking, embodiment can be an effective approach to assist the process.

Three papers (7%) in our sample explored the use of embodied cognition to enhance debugging skills [4, 121, 130]. Ahn et al. investigated two forms of embodiment for a maze traversal task, where elementary-school students needed to debug a character programmed to follow a certain path [4]. The first involved direct embodiment, where students physically walked through the maze, while the second utilized surrogate embodiment, with students manipulating a paper character through the path. Both of these approaches were designed to help students embody the association between a command block and the resulting outcome. Another effective form of embodiment is through unplugged programming activities. These methods offer concrete and content-focused learning experiences for novices, reducing the cognitive demands associated with technology and technical knowledge. To illustrate, Sipitakiat et al. (2012) created the Robo-blocks system, in which children control the movement of a floor robot by snapping together physical command blocks [130]. These blocks were easily rearrangeable, enabling students to promptly adapt them in response to errors. The designers also included a step-by-step function, which let children slow down the execution process to observe each command block and its associated action.

Metacognition. Debugging is rarely a linear process and thus requires students to metacognitively monitor their progression and thinking throughout. In addition to applying their debugging knowledge, students must also learn to self-regulate their emotions, strategy usage, and pacing [150]. Studies find that during programming writ-large, students often employ variable metacognitive strategies and self-regulate in shallow and unsuccessful ways [20, 49]. Encouragingly though, several papers confirm that students can be taught metacognition skills and that these skills improve their coding ability [21, 92]. Prior interventions on programming metacognition have aimed to target self-regulation before, during, and/or after the problem-solving process, in accordance with self-regulation theory [93].

In our sample, which included 16 (37%) metacognition interventions, we found that papers similarly targeted each of these stages with their interventions. DeLiema et al. (2019) encouraged students to write "tweets" *before* coding, as reminders to their future selves [36]. Students referenced these strategies, such as "#becalm," "#think about it," and "#trydifferentstrategies," whenever debugging became difficult. Other interventions in our sample scaffolded the metacognitive process *while* students were debugging [1] (Figure 4). For instance, Ko et al. (2019) built a tool that walked students through the process of systematically locating a defect. Using this tool, students located where they were in the debugging process and could track their own progress [80]. Another common approach was to constrain the progression of the debugging process—students must first accurately identify the bug location before proceeding to solve it. This was implemented in both online and unplugged activities, either with virtual [95] (Figure 4) or physical bug flags [130]. Lastly, researchers employed reflection activities *after* the debugging experience, to help students consolidate their strategies or process their emotions. Reflection was typically written [24, 42, 91] or verbal [141, 144], though Dahn et al. (2020) incorporated arts instruction alongside the programming instruction to help students reflect on their debugging emotions through various art forms [34].

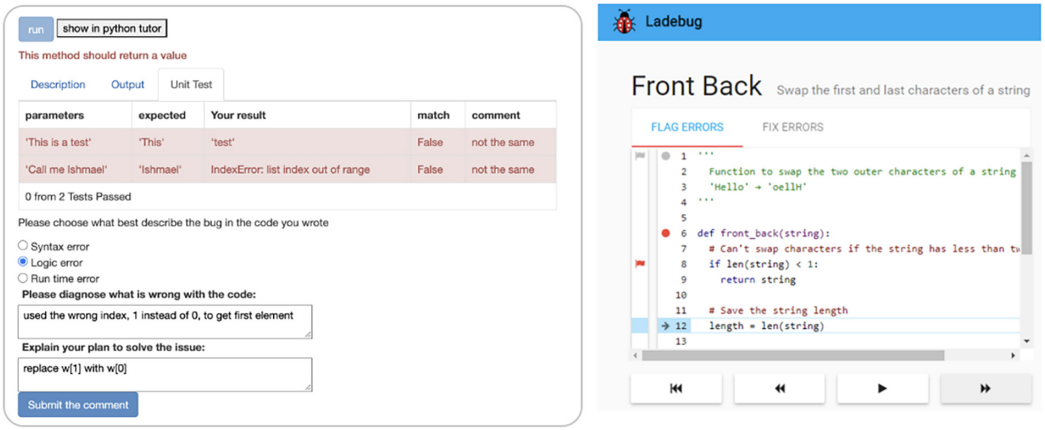

Fig. 4. *Examples of metacognitive scaffolding interventions.* The example on the left (screenshot from [1]) depicts the programming environment, Spinoza. As learners solve a series of coding problems, the debugging process is metacognitively scaffolded for them. When they encounter a bug, students must first diagnose the error, then create a plan to solve the issue. The example on the right (screenshot from [95]) depicts the coding environment, Ladebug. Students work through a series of debugging exercises and must first correctly locate the error before fixing the bug.

Observation. Expert debuggers often find it difficult to describe their debugging process since their expertise relies on pattern recognition that commonly occurs subconsciously [132]. Because of this, *demonstrating* debugging practices is often more effective than verbal instruction alone. In an early line of work, researchers experimented with replaying the eye gaze patterns of experts while debugging, also known as eye movement modeling examples [135]. This form of modeling can enhance students' understanding of code, since eye gaze conveys the intricacies of visual processing strategies [19]. While social learning theory emphasizes the role of teaching by modeling ideal procedures [14], cognitive load theory also highlights the use of worked examples in observational learning [137, 138]. In contrast with learning by problem-solving, where students may fixate on correctness, studying worked examples enables them to focus solely on learning the ideal procedure for solving such problems [139].

Within our sample, which included 9 (21%) examples of observational learning, interventions implemented both modeling [24, 36, 80, 104, 119] and worked examples [22, 30, 143, 144]. Modeling typically took place in a live programming context, where instructors coded for students in front of the class or in one-on-one sessions. In these interventions, the instructor either demonstrated a programming exercise and debugged as errors arose [119], or modeled a specific debugging strategy, such as iterative hypothesis testing [24]. Instructors in the study by DeLiema et al. (2019) implemented modeling behaviors in smaller groups and one-to-one [36]. Informed by the tenets of reciprocal teaching, they modeled strategies while narrating what an expert might think when enacting them. Along the way, they also prompted students to reflect on how that strategy worked and contributed to debugging. Bofferding et al. investigated how analyzing worked examples of debugging could contribute to student learning [22]. They presented first- and third-graders with correct and incorrect worked examples of common bugs in a block-based programming language. These examples included explanation prompts and asked student pairs to identify why certain code was used, what it did, or what the bug in the program was. Students were then asked to apply these procedures to an example exercise. The authors hypothesized that these worked examples could help students internalize the procedures of effective debugging strategies.

Scaffolding. Scaffolding describes the material and social supports that help learners begin to participate in an authentic activity. In the beginning, these structures help to reduce the cognitive burden of a complex task and are gradually reduced as the student gains proficiency and expertise [142]. A plethora of programming tools have aimed to scaffold or simplify the debugging process. Most notably, block-based languages were designed to eliminate syntax errors so students could focus on the act of creating and designing [120]. Specific to debugging, researchers have built tools to help novices interpret error messages, either by rewriting them [18, 37] or suggesting solutions that peers have applied in the past [64]. Still, the predominant objective of these tools is often not for students to learn debugging, per se, but to streamline the process so students can focus on other programming goals [64]. Our review and categorizations attempted to describe scaffolding mechanisms that authors themselves acknowledged as aiding debugging *instruction*. This is an understandably broad category, which overlaps with many other pedagogical approaches described above.

Among the papers that we reviewed, which included 28 (65%) examples of scaffolding, intervention designs broadly fell into four categories: metacognitive scaffolding, simplification, constraints, and hints. The debugging process can be *metacognitively scaffolded* by providing a formalized procedure for students to follow [47, 80, 95, 104, 141, 143, 144]. For example, Abu Deeb and Hickey (2021) piloted Spinoza, a reflective debugging platform, in an introductory CS course [1]. As students encountered errors while solving programming exercises, the software prompted them to identify the type of bug, diagnose what was wrong with the code, and explain their plan to solve the issue (Figure 4). Debugging can also be overwhelming, and *simplifying* tasks reduces cognitive load. Exercises can be designed to have only one error or use one strategy, to help students focus on practicing specific bug types or skills [71, 105]. Unplugged exercises can help students focus on learning the debugging process and strategy over the tools and technology [4, 130]. *Constraints* within a debugging environment can also be helpful to limit maladaptive strategies. For example, a handful of exercise-based interventions limited the lines of code that learners could edit, either to help them narrow down on suspect code [51] or to prevent them from deleting large chunks of code or introducing new errors while debugging [91, 95]. Lastly, intervention designs included pre-written hints or error messages to provide clues to the error location or bug fix [3, 10, 28, 46, 58, 67, 136].

4.3 Debugging Process

Although debugging is often conceptualized as a single act, it requires students to master and apply several sub-skills [75]. In this section, we consider our second research question by examining which steps of the debugging process, as summarized by Li et al. (2019) [88], are targeted by the various interventions in our sample. In the following sections, we describe each step of the debugging process, why novices may struggle at this stage, and summarize studies in our sample that specifically targeted this phase. Table 5 shows an overview of how interventions targeted each step in the debugging process with paper references.

Step 1: Construct the Problem Space. The first step in the debugging process is to construct a mental model of the system and the code. This summarizes the intended behavior or state of the program, the actual behavior, the function and structure of the program, and finally the execution and control of the program [88]. Ideal strategies that support this step are understanding the programming language [53, 60], understanding the code [78, 140, 148], and program tracing [108]. A robust understanding of the code allows students to debug more efficiently and strategically [60]. However, this is also the step that novices most frequently forego [129]. When faced with an error, beginners will often jump to testing hypotheses without taking time to understand the program

Table 5. Summary of How Interventions Targeted Each Step in the Debugging Process and Non-Cognitive Debugging Skills

Debugging Step	Methods to support this step from sample	Count
Step 1: Construct the problem space	— Visualizations of program execution [10, 32, 33, 35, 67, 136] — Teaching comprehension strategies, such as program-slicing [47], code-tracing [105], print statements [105], and writing out the code in natural language [54] — Answering comprehension questions about the code [22] — Acting out the code with embodiment [4]	11 (26%)
Step 2: Identify fault symptoms	— Pre-made test cases for student code [1, 24, 51, 67, 95, 97, 143, 144] — Hints or re-written error messages [3, 28, 71] — Teaching explicit testing processes [80, 141] or strategies, such as identifying boundary conditions [54] and comparing the expected and actual output [51, 104, 143, 144] — Visualizations comparing expected and actual output [136]	18 (42%)
Step 3: Diagnose the fault	— Explicit instruction and practice on bug location strategies [36], such as print statements [51, 54, 105], decomposition [105], hypothesis testing [24, 80, 143, 144], and diagramming the code [54] — Debuggers to assist tracing [67, 91, 95, 130, 136] — Hints and clues to fault location [3, 10, 46, 58, 71, 136] — Identifying stereotypical bug patterns [30, 36, 52, 146]	24 (56%)
Step 4: Generate and verify solutions	— Creating a plan to solve the bug [1, 36, 143, 144] — Specific hints for the correct solution [3, 58, 71, 85, 136] — Protecting original source code so new errors aren't introduced during the fixing process [51, 91, 95] — Pre-made test cases for student code [1, 24, 51, 67, 95, 97, 143, 144] — Students write their own test cases [42]	17 (40%)
Step 5: Reflect and Document	— Class discussions to identify stereotypical bug patterns [34, 36, 52, 141] — Writing self-explanations of the bug cause [91] and fix [42] — Written, verbal or visual reflections on debugging strategies [36, 144] or debugging process [24, 34, 144]	9 (21%)
Non-cognitive		
Self-efficacy	— Scaffolding bug difficulty to build up confidence [4, 67] — Practicing common bugs and debugging strategies [51, 52, 80, 104]	6 (14%)
Mindset	— Attributing errors to the computer's misunderstanding [46, 85] — Storytelling, sharing and reflecting on bug encounters [34, 36, 42, 52]	5 (12%)
Affect	— Collaborative sharing of debugging struggles [34, 36, 52, 107] — Creating art to process debugging emotions [34, 36] — Instruction on emotional awareness [54]	5 (12%)
Persistence	— Metacognitive scaffolding [1] — Gamification and narrative to motivate debugging [35, 71, 85, 105]	5 (12%)

Article counts are presented in the right-most column.

first [108, 140]. This step is also challenging since novices are operating with a fragile knowledge of program execution, making it difficult to trace dependencies and reason backwards from the error [116].

In our sample, 11 papers (26%) included interventions that supported this step in the debugging process. One common approach was to directly teach students code tracing and comprehension strategies [47, 54, 105]. Eranki et al. introduced students to a professional debugging technique: program-slicing. Students were taught to decompose code semantically through exercises correcting jumbled code and identifying missing lines of code [47]. These exercises were designed to help students locate code dependencies to assist the debugging process. Another approach was to ask students code comprehension questions to confirm their understanding of the original code. In their worked examples, Bofferding et al. included reflection prompts such as "which coding piece tells Awbie to do Steps 2 and 3 of the program?" These discussion questions prompted students to check their own understanding, allowing them to better reason about why the example code was or was not working [22]. Ahn et al. (2022) [4] also explored the use of embodiment, where students act out the code, to help them gain a better understanding before debugging. Similarly, visualizations of variable states, program execution, and algorithms aimed to assist students in building a mental model of the code. These tools externalized an overview of the program and sought to help students internalize the process of code tracing. Albeit, visualizations are a passive way of supporting this step, since prior work has shown that students may not actively engage with visual aids unless prompted to through engagement prompts [133].

Step 2: Identify Fault Symptoms. Upon pinpointing the intended and actual program behaviors in the first step, the next task is to identify any inconsistencies between these two states. The strategies employed here vary depending on the nature of the bug, whether it manifests at compile-time, runtime, or stems from a logical error. When dealing with a compilation error, a valuable technique is to notice and interpret the error message [74], which contradicts the intended program behavior. Bugs occurring at runtime may not always manifest, so thoroughly testing the system is a useful strategy to pinpoint states which are abnormal or cause the program to crash [40, 140]. Lastly, when searching for logical errors, it can be helpful to clarify how the output differs from the intended behavior, or map out instances where the code outputs correctly and incorrectly [29, 78]. Strategies to support this step are commonly challenging for novices since error messages are notoriously difficult to understand [18, 37], and students often lack the procedural knowledge to thoroughly test their code [108]. Additionally, beginners generally struggle to identify what the code is supposed to do and have trouble detecting discrepancies between the intended and actual behavior [2, 60].

In our sample, 18 (42%) papers supported this step in the debugging process, using approaches such as rewriting error messages [3, 28, 71], offering explicit guidance on testing code [54, 80, 104, 141, 143, 144], asking metacognitive questions [141, 143, 144], and providing pre-written test cases [1, 24, 51, 67, 95, 97, 143, 144]. To help students comprehend and address error messages, Carter (2015) and Jemmali (2022) embedded hints within their programming environment that elucidated the compiler error [28, 71]. Five interventions explicitly taught students to test their code using direct instruction or exercises. For example, in their debugging unit, Whalley et al. [143] included a specific lab session teaching students testing strategies. Students were shown an example of faulty code that passed and failed a series of test cases. They were then asked to compare the difference between the failing test case's output and the expected output, honing their skills of locating discrepancies. To teach students to reason through logical errors, Fenwick created a debugging exercise called Quicksand [51]. In this activity, the original source code is hidden from students, and they must locate the error by observing which test cases the code passes and fails. Interventions situated in the computational thinking literature also employed metacognitive

prompting [25]. Following a debugging session, students were asked to describe what happened when they ran their code, and how it differed from what they wanted and expected [141]. These questions are designed to help students internalize a mindset of testing and comparing. The most prevalent approach to help students identify fault symptoms was to provide a pre-made set of test cases. Yet, the majority of these approaches only require passive engagement from the students, leaving it unclear as to whether students internalized how to generate test cases or the importance of it. Notably though, Bottcher et al. (2016) intentionally included erroneous test-cases in their sets, encouraging students to think critically about crafting meaningful test-cases [24].

Step 3: Diagnose the Fault. After discerning disparities between the program's current state and its intended behavior, the subsequent phase is to diagnose the fault and pinpoint the bug's location. Vessey suggests that experts often take an approach akin to the scientific method—generating a series of hypotheses and iteratively testing each one until the error is exposed [140]. Common techniques to optimize this stage often require strategic and experiential knowledge [88]. Programmers rely on prior experience with similar bugs to hypothesize likely causes and use a series of strategies to narrow down on the correct origin of the bug [40]. Due to inexperience and/or ineffective use of strategies, students new to programming struggle at this stage. Murphy et al. (2008) noted that they often employ maladaptive strategies, such as trial and error, or deleting the entire code [108]. Further, novices may over-focus on a single hypothesis instead of iteratively considering alternative causes [53, 73, 140].

Of the papers that we reviewed, 24 interventions supported this step, around 56% of our sample. The intervention designs can be categorized into a few distinct approaches, which include: teaching hypothesis generation and specific debugging strategies [24, 36, 51, 54, 80, 105, 143, 144], providing hints to the error location [3, 10, 46, 58, 71, 136], designing visual debuggers to help students step through the code and locate the fault [67, 91, 95, 130, 136], and helping students to consolidate and chunk stereotypical bug patterns [30, 36, 52, 146]. A number of papers highlighted the importance of hypothesis generation, with accompanying visuals or process illustrations to highlight the iterative nature of hypothesis testing. For example, in their debug strategy, Ko et al. taught students to flag suspect code that might be causing the incorrect behavior, then to loop through each one until they located the fault [80]. Michaeli and Romeike (2019) also shared a visual aid of the debugging process with students, which situated "hypothesizing about the cause" in repeated loops [104]. Interventions also taught students explicit debugging strategies, such as adding print statements, decomposition and using a debugger. Debuggers that enable students to step through their code prove valuable in supporting this phase, since they offer a systematic approach to examine the code's execution and pinpoint the error's location. Another common approach, most often embedded in programming environments, was to provide visual or textual hints to the error location. For example, Edmison and Edwards (2020) used SFL to overlay heatmap visualizations on student's code [46]. The colors of the heatmap provided visual clues to the bug's most likely location, which was designed to prevent students from going down the wrong path. Lastly, to build up students' experiential knowledge of common bug patterns, Fields (2021) and Deliema (2019) incorporated class discussions into their debugging curriculum [36, 52]. Students brainstormed common bugs that they ran into and collaboratively categorized these as a class. This form of recounting and chunking was crafted to help students recognize patterns among errors to aid the next time they encountered similar bugs.

Step 4: Generate and Verify Solutions. Once the bug has been located, the final step is to generate and verify solutions to the code. Interestingly, the primary challenge in debugging does not lie in solving the bug, but rather identifying it. Novices who were able to correctly locate the bug almost always solved it [53, 74]. After resolving the bug, programmers should also reevaluate the program, verifying that the fix corrected the initial issue without inadvertently introducing new

errors. This step is one that students frequently overlook—Fitzgerald et al. noted that students rarely test their code with boundary conditions, and also tend to introduce more errors during the debugging process [53].

In our sample, 17 (40%) interventions specifically supported students in generating and verifying their solutions, primarily through metacognitive scaffolding techniques [1, 36, 143, 144], hints [3, 58, 71, 85, 136], and pre-made test cases [1, 24, 51, 67, 95, 97, 143, 144]. To help students generate solutions, the online IDE, Spinoza, tasks students with writing a plan about how they will solve the bug after diagnosing the error [1]. Specific hints, such as "try; instead of)" [1], and relevant examples of correct syntax [3] also lead students to the correct solution of the bug. The programming tools Ladebug [95] and Quicksand [51] protect the original source code once students have located the bug, so that students don't introduce more errors when fixing the original bug. Lastly, pre-made test cases were designed to help students verify their solutions, similar to their usage in Step 2. Instructors often included boundary conditions in these tests to encourage students to think about edge cases. Duwe et al. specifically set out to teach students a "testing mindset," which they define to be the belief that "if [the code] wasn't tested, it doesn't work" [42]. As students worked on their electronic projects, they collaboratively built up test sets for their processors and assigned each member to test submodules of the tool. These activities were crafted to reinforce the importance of testing, even in the post-debugging phase.

Step 5: Reflect and Document. The last step in debugging is to reflect on the overall process. Professionals frequently keep bug logs [79, 117] to document common errors for future reference, and reflection can help students consolidate their knowledge. This aligns with a central Deweyan philosophy that learning does not result from experience alone, but rather *reflection* on experience [122]. Fitzgerald et al. observed that when debugging, a subset of novices just "stumble[d] upon [the solution] haphazardly," without understanding what the bug was or why their fix worked [53]. A final step of reflection after solving the bug can help students transform their experience into expertise.

In our sample 9 (21%) papers actively promoted reflection after debugging. The primary aspects that students reflected on were stereotypical bugs [34, 36, 52, 141], details about the bug [1, 36, 42, 91], the effectiveness of certain debugging strategies and processes [24, 36, 144], and their emotions [34]. To highlight common bugs, Fields et al. (2021) asked students to discuss frequent errors they encountered [52]. As a class, they categorized these problems into groups and displayed them on posters around the classroom. In their debugging game, BOTS, Liu et al. (2017) included a self-explanation feature to encourage reflective thinking [91]. After solving the bug, the system prompted students to update their initial hypothesis of what the error was. DeLiema et al. (2019) asked students to reflect on effective strategies for locating errors, and helped them express the emotional experience of debugging through painting, drawing comic-strips, data visualizations and writing poems [36]. After teaching students a formal process for debugging, Whalley et al. (2021) conducted debugging sessions with a handful of students and asked them to reflect on their process and whether they would make any change next time [144]. Reflection was primarily verbal or written (e.g., in bug journals), though researchers have also explored the use of artistic mediums [34]. It commonly took place collaboratively, either between pairs or in a classroom discussion.

4.4 Non-Cognitive Skills

To answer our third research question, our review also investigated how/if studies addressed the non-cognitive skills related to debugging. Sixteen papers (43% of our sample) mentioned how their approach may impact these skills, or measured a non-cognitive construct. Self-efficacy (six papers) was most frequently studied, followed by mindset (five papers), affect (five papers), and

persistence (five papers). These constructs often overlap and interact, leading to instances where papers simultaneously address multiple constructs. We describe each construct and summarize how studies in our sample targeted these. A summary of these approaches, along with references, can be found in Table 5.

Self-Efficacy. Self-efficacy is a psychological construct describing the beliefs that an individual holds about what they are capable of accomplishing [12]. These beliefs develop continuously, influenced by external factors, such as encouragement, and internal factors, such as emotions and self-assessments of performance [13]. Although self-efficacy and related interventions have been studied extensively in CS education, less is known about students' perceptions of their debugging ability specifically. Improving students' debugging self-efficacy is important since these beliefs influence behavioral outcomes essential for debugging. For example, students with higher self-efficacy exhibit greater effort and persistence [6, 114] and are more likely to employ self-regulated behaviors, such as systematic thinking and strategy usage [124].

One of the common approaches for bolstering self-efficacy was using scaffolding techniques such as incrementally adjusting the difficulty of the bug or presenting the program in natural language before proceeding to code [4, 67]. By starting with manageable challenges and progressively introducing more intricate bug scenarios, the intent is to help learners feel less overwhelmed and more confident in their debugging skills. Another prevalent strategy involved deliberate practice, where students engaged in focused, purposeful practice targeting specific types of bugs or debugging strategies [51, 80]. This approach aimed to cultivate a sense of mastery and confidence in students' ability to tackle similar bugs in the future.

Mindset. Researchers have suggested that there are certain mindsets associated with debugging. One mindset in particular is a growth mindset. Students with a growth mindset believe intelligence is a malleable trait [43, 44]. They prioritize learning over performance and show persistence in the face of challenges [45]. This mindset is especially relevant when debugging because bugs are inherent challenges and errors. Students with a growth mindset though may see these obstacles as learning opportunities rather than a sign of failure [111]. In addition to having a growth mindset, Duwe (2022) suggests that students must also develop the mindset that bugs are an expected part of the development process and that debugging requires a systematic approach [42]. These mindsets are important because they often dictate how students respond in the face of challenging errors.

One approach to address students' mindsets was through reframing techniques rooted in attribution theory. Lee et al. (2011) intentionally created a game where the compiler blames itself for errors—reframing bugs as a misunderstanding rather than a failure of the student [85]. As part of their curriculum, Duwe et al. (2022) asked students to complete "debugging demos" where they demonstrated how a bug could illuminate a learning concept [42]. Instructors and teaching assistants were also encouraged to model a debugging mindset during class and labs. Another common approach was to engage students in storytelling and sharing, to frame bugs as a common occurrence and learning experience [34, 36, 42, 52]. Fields et al. had students intentionally create buggy code for their peers to solve, reinforcing the belief that bugs are a natural part of the coding process and a shared experience [52].

Affect. When debugging, students often experience a host of negative emotions, such as frustration and stress [18]. While mild levels of confusion and frustration are important for effortful learning [11, 102] and debugging [99], intense and prolonged negative affect can lead to despair [125] and disengagement [39]. In the later state, students often resort to systematic guessing [94] or "gaming the system" behaviors [5], both of which are unproductive for effective debugging. Additionally, research has found that in CS courses, early experiences of frustration can impact

student learning outcomes on later projects [90]. Thus, it is important to identify interventions that can help students regulate their emotions while debugging.

Dahn et al. (2020) and Deliema et al. (2019) helped students reframe and understand their debugging emotions by creating and sharing works of art to illustrate their feelings. Processing moments of failure externally helps students rethink how they dealt with and channeled their emotions, prompting deeper reflection and regulation [34, 36]. In a debugging manual provided to students, Garcia et al. (2022) included information about emotional awareness to help students recognize how their negative emotions could influence their problem-solving process [54]. Lastly, sharing experiences of struggle can support empathy among peers. Murphy et al. (2010) point out that pair programming can help process and vent the negative emotions associated with debugging [107].

Persistence. Many of the non-cognitive constructs described above manifest behaviorally in the amount of effort and persistence students invest in overcoming bugs. Debugging is an inherently time-consuming and difficult process—expert programmers report spending over 50% of their working hours fixing bugs [62]. It is a skill that requires persistence, which is the commitment to tackle a problem for long durations of time, even in the face of obstacles. While persistence is typically conceptualized as a positive quality that is important for learning [41, 128], researchers have also noted that there may be unproductive forms of persistence. For example, an important self-regulatory strategy is to evaluate when one is truly stuck and needs help, rather than persisting with little progress and learning [5]. A challenge in debugging instruction is to help students persist through difficult bugs, while also equipping them with effective help-seeking strategies to avoid unproductive "wheel spinning" [16].

To target debugging persistence specifically, researchers have explored scaffolding and gamification techniques. Jemmali et al. (2022) created personalized bugs based on prior knowledge, hoping to gradually increase the difficulty level or errors students faced [71]. To reduce cognitive load and break down larger debugging problems, Abu deeb and Hickey (2021) metacognitively scaffolded the debugging process [1]. Additionally, Lee et al. (2011), Miljanovic and Bradbury (2017), and Deitz et al. (2016) couched debugging in fun games and compelling narratives, aiming to increase motivation and persistence when solving bugs [35, 85, 105].

4.5 Evaluation Outcome and Effectiveness

While the previous sections summarize the design of the interventions in our sample, this section describes the constructs used to measure their effectiveness and the resulting outcome. This is in service of answering our fourth research question, which considers what methods are used to evaluate the interventions in our sample, and our fifth research question, which explores what is known about the effectiveness of these interventions. First, we summarize the experimental design of the studies. Next, we summarize the evaluation methods used to assess different debugging constructs in Table 6. In the sections below, we summarized papers by the debugging construct they assessed and reported qualitative and quantitative findings from interventions' assessments. Figure 5 summarizes the results of interventions that utilized significance testing, organized by debugging construct and assessment method.

All but two of the papers in our sample evaluated their intervention along some metric. The study designs in our analysis range from non-experimental (53%, 23 papers), to quasi-experimental (16%, 7 papers), to experimental (33%, 14 papers). Interventions were most commonly assessed through observational techniques, including both quantitative (e.g., counts of how many bugs students solved during the task) and qualitative (e.g., instructor observations) measures. Other assessment methods included interviews, surveys, written reflections, and learning assessments.

Accuracy. In our sample, papers most commonly assessed the effectiveness of their interventions by measuring debugging accuracy (21 papers, 49%). Of the papers that accessed debugging accuracy, 18 (86%) conducted significance testing, either through experimental (12 papers), quasi-experimental (3 papers), or non-experimental (3 papers) setups. The results were largely successful, with 12 papers [4, 8, 10, 22, 30, 32, 33, 47, 58, 104, 131, 146] finding significant improvements on some measure of accuracy. Among university and adult learners, deliberate practice and visualizations were effective. Direct instruction, practice, and modeling within workshop contexts were helpful for K-12 students.

Additionally, papers that conducted learning assessments showed that these interventions could help students learn to debug better [4, 8, 10, 22, 30, 47, 103, 131, 146]. For example, Miljanovic and Bradbury (2017) found that after playing the debugging game, Robobug [105], university students performed significantly higher on a learning test about debugging strategies. Ahn et al. (2022) observed that elementary-school students who received an embodiment intervention tested better than the control group who did not [4]. However, two papers found that the benefits of their intervention did not transfer when the intervention scaffolds were removed [58, 149]. For example, adult participants in the study by Greifenstein et al. (2021) were able to correctly fix more bugs when they were provided with hints within their coding environment. However, without hints, the experimental group did not perform better than the control [58]. Viewed as a whole though, these summarized results suggest that interventions can improve debugging accuracy in both K-12 and university settings.

Efficiency. Another measure used to assess interventions was how long or how many steps it took students to debug—debugging efficiency (13 papers, 30%). Among papers measuring efficiency, 10 papers (77%) conducted significance testing, using experimental (8 papers), quasi-experimental (2 papers), and non-experimental designs (1 paper). Interventions were largely successful, with eight papers reporting significant improvements in debugging efficiency [1, 3, 8, 10, 28, 32, 33, 58], primarily among university and adult learners. For instance, Abu Deeb and Hickey (2021) found that prompting university students to reflect on the bug and plan their solution reduced the number of runs needed to solve the error [1]. Alrashidi et al. (2017) also showed that when university students were given an augmented view of the problem space through AR, they spent less time locating and fixing the bug [8]. Still, these results did not always transfer to *learning*. Four articles measured students' debugging efficiency without the intervention, all of which found no effect [3, 10, 58, 149]. For example, Ahmed et al. (2020) found that while embedded syntax hints and examples helped students correct their errors more quickly and with fewer attempts, they did not subsequently debug faster without these hints [3]. Collectively, these results suggest that while interventions may be effective at improving efficiency, these improvements may not always transfer when the scaffolds are removed.

Adoption. Researchers also assessed the extent to which students adopted the debugging strategies they were taught (11 papers, 26%). Only one paper conducted significance testing on adoption, showing mixed results [141]. When evaluating the impact of direct instruction on debugging principles, Vourletsis et al. (2021) found that middle-school students used more systematic debugging strategies during the last unit of their course, but that their usage did not consistently increase across all units. Qualitative reports from other studies generally found that students struggle to apply learned strategies when debugging their own code. For example, Böttcher et al. (2016) calculated that "roughly half" of the university students who were taught a systematic approach to debugging had difficulty applying this to their lab exercises [24]. Many resorted to "random[ly] poking around." Similarly, Ko et al. (2019) reported that secondary school students struggled to regulate their strategy usage, often "defaulting to ineffective trial and error methods, even when

they knew systematic strategies would be more effective" [80]. Whalley et al. (2021) noted similar findings and also highlighted that only a small fraction of students reflected that their process was flawed and expressed interest in learning a more formal approach [144]. Taken together, these results suggest that interventions struggle to improve adoption of debugging strategies, both among K-12 and university learners.

Perceived Helpfulness. Seventeen papers (40%) measured students' perceptions of the helpfulness of the intervention, primarily through surveys, interviews, or written reflections. Two papers conducted statistical testing on perceived helpfulness [71, 136]. Jemmali et al. (2022) found that students who received personalized bugs based on their previous errors rated the game to be more helpful for their learning than the control alternative [71]. Students in Suzuki et al. (2017) rated visualizations comparing code traces of the actual and intended output of the code as helpful for identifying and understanding the bug, but did not perceive that it improved their debugging skills [136]. Papers that assessed the usability of their tool generally reported "good" and "acceptable" results based on standardized usability scales [47, 95]. Qualitative findings provided more nuanced insight into how interventions impacted student debugging. For example, when university students were presented with a debugging manual of strategies, one student reported that it "gave new methods to understand how to debug" and "think about different approaches to debugging" [54]. When asked to provide open-feedback on a live-coding session, students reported that it helped them understand the possible bugs that could occur while coding [119].

Self-Efficacy. As a primary non-cognitive outcome, six papers (14%) in our sample assessed students' self-efficacy after the debugging intervention. Three studies in our sample reported statistical significance testing for self-efficacy assessments, all for K-12 students [4, 104, 149]. Systematically teaching students debugging strategies in a short workshop format can increase their self-efficacy [104] and embodiment of the code can similarly help [4]. However, Zhong and Li (2020) reported that students debugging in pairs did not differ significantly in their self-efficacy ratings compared to students who worked individually [149]. Thus, while direct instruction and embodiment can improve debugging self-efficacy among K-12 students, collaboration may not be an effective approach.

Mindset. In our sample, four papers (9%) assessed debugging mindset. Three papers assessed mindset among middle- and high-school students [34, 36, 52], and one paper assessed it among university students [42]. Although no studies conducted significance testing on mindset, qualitative results suggest that students began to view bugs as learning opportunities and an expected by-product of programming. Effortful reflection proved to be a catalyst for mindset shifts [34, 42]. For example, one student from Dahn et al. (2020) reflected at the end of their workshop that "I don't view bugs as a bad thing, but instead as a positive, to improve." Duwe et al. (2022) also observed that students began to see bugs as an opportunity to "connect the dots" and identify gaps in their knowledge to learn more [42]. Additionally, when students designed bugs for each other to solve and shared about their debugging struggles, they realized that "a lot of people make mistakes" [52]. While additional studies should substantiate these results, the findings present encouraging preliminary evidence that interventions encouraging reflection can help students foster a debugging mindset.

Affect. Six papers (14%) in our sample assessed student affect. Two were conducted with university students in a lab setting [105, 144], and the other four were with K-12 students during week-long workshops [34, 36, 52, 131]. Two papers conducted statistical testing on affect [105, 131]. Miljanovic and Bradbury (2017) measured student emotions before and after playing their debugging game, Robobug, and found a non-statistically significant *decrease* in students' positive affect and *increase*

Table 6. Summary of Intervention Evaluations

Construct	Method	Metric
Accuracy ($n = 21$)	Observational	—# of debugging exercises solved [1, 32, 35, 107] —# of errors fixed [33, 58, 71, 136, 149] —# of errors encountered when programming [3, 33, 71, 131, 149] —# of errors introduced when debugging existing errors [33] —Expert graded score for debugging problem [10]
	Learning assessment	—# of questions correct on debugging test [4, 8, 22, 28, 30, 47, 104, 105, 131, 146] —Expert graded score for debugging problem or project [10, 149] —# of errors fixed (without intervention) [58, 71]
Efficiency ($n = 13$)	Observational	—Total time solving bug or exercise [1, 3, 8, 10, 33, 46, 58, 91, 107, 136, 149] —# of runs / attempts to complete the debugging exercise [1, 3, 28, 91] —# of code edits (effective, ineffective or neutral) to debug [32, 91] —# of times students solved bug on first attempt [91]
	Learning assessment	—Time spent on learning test or test project [149] —Total time solving the bug or exercise (without intervention) [3, 10, 58] —# of runs to complete the debugging exercise (without intervention) [3]
Adoption ($n = 11$)	Observational	—Experimenter observations of strategy usage [22, 42, 91, 130] —Qualitative analysis of in-class help requests [80] —Log data from strategy usage tracker [80] —Log data from program submissions [47] —Think-alouds while debugging [141]
	Written reflection	—Coding students' documentation of their debugging process [24, 42, 141, 143]
	Survey	—Self-report of whether students used debugging strategies [80]
	Interview	—Asking students about their debugging process and strategies [36, 141, 144]
Perceived helpfulness ($n = 17$)	Observational	—Researcher observing student behavior and interactions with the intervention [34, 36, 52, 80, 107]
	Written reflection	—Student reflections coded for perceived helpfulness [34, 36, 52]
	Survey	—Likert surveys asking about the interventions usability [33, 47, 54, 71, 95] and impact on learning or debugging [33, 35, 58, 91, 136] —Open ended feedback about the intervention [80, 95, 119]
	Interview	—Interviewing students about their experience with the intervention [34, 36, 52, 80, 130, 144]
Self-efficacy ($n = 6$)	Survey	—Researcher-developed self-efficacy survey [36, 51, 80, 104] —Adapted validated self-efficacy survey [4, 149]
Mindset ($n = 4$)	Observational	—Instructor observations of student behavior [42]
	Written reflection	—Student reflections coded for indications of mindset shift [42, 52]
	Interview	—Student talk indicating a debugging mindset [34, 36, 52]
Affect ($n = 6$)	Written reflection	—Students' reflections coded for emotions about debugging [34, 36]
	Survey	—Adapted validated affect surveys [105, 131]
	Interview	—Interviews capturing how students talked about emotions in the context of debugging and coding [34, 36, 52, 144] —Capturing student sentiment towards the intervention [52]
Persistence ($n = 3$)	Observational	—# of people who gave up on the coding exercise [1] —# of runs before giving up [1] —# of game levels attempted [71, 85] —Time spent playing game [71, 85]
	Learning assessment	—# of game levels attempted (without intervention) [71] —Time spent playing game (without intervention) [71]

This describes how studies assessed their intervention. Approaches are summarized by debugging construct, assessment method, and specific metric, with accompanying article citations for reference.

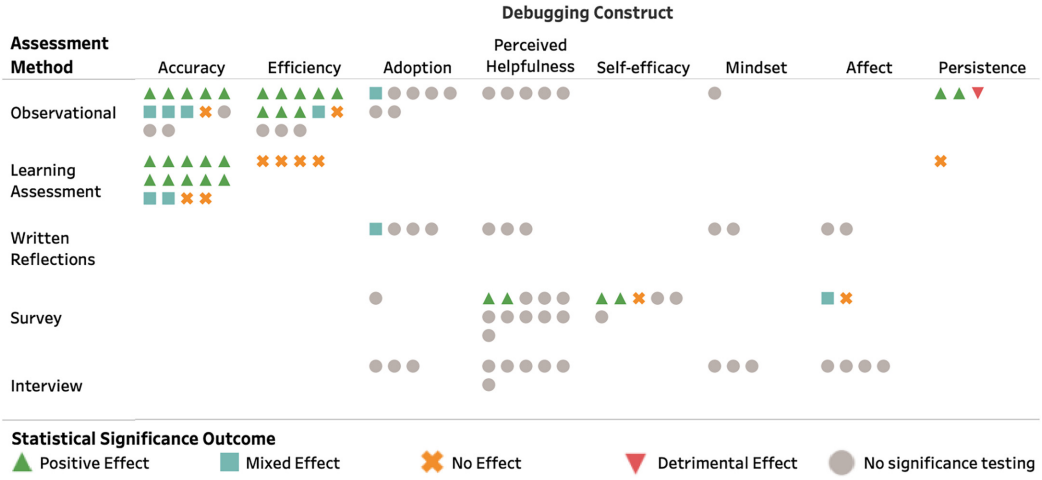

Fig. 5. *Summary of intervention effectiveness by assessment method and debugging construct.* This figure presents the outcomes of interventions assessed through various methods against different debugging constructs. Each marker represents an intervention's result, with the color and shape indicating the statistical significance outcome. If a paper did not assess statistical results, it is presented as a gray circle. A single article may appear in a column multiple times if it used different assessment methods to evaluate the same construct. The position and color of the markers under each debugging construct provide insight into the most commonly used assessment methods and their evaluation result. For example, adoption, perceived helpfulness, and mindset were infrequently assessed with significance testing. While measures of debugging efficiency are largely successful when measured with observational measures, measures using learning assessments show no effect. Lastly, statistical results for affect and persistence show mixed results. Detailed results with citations are presented in the text.

in negative affect. The authors speculate that the game mechanics led to some user frustrations, but it was unclear whether the game helped students feel less anxious or frustrated when debugging their own code. Socratous et al. (2020) compared a structured (direct instruction) and unstructured (exploratory) approach to instruction for block-based programming. While students in the structured group learned more debugging skills, students in the unstructured group reported significantly higher emotional engagement with the task [131]. These findings suggest that an open-ended approach to programming instruction may encourage more positive emotions for elementary-school students. Qualitative results from Dahn et al. (2020) showed that art-making can help students reframe their emotions while debugging [34]. Although students still described intense reactions to bugs such as "wanting to cry," they also described important realizations of regulation, such as "how nervousness can…become something beautiful if you change the perspective on it." Fields et al. (2021) [52] and Deliema et al. (2019) [36] describe similar findings among secondary school students after week-long workshops with intentional emotional scaffolds. Collectively, these studies suggest that the usability of a debugging interface can increase distress, but that *open-ended* and *long-term* interventions normalizing and reframing debugging may reduce frustration and stress among K-12 students.

Persistence. In our sample, three papers explored measures of persistence while debugging, all using process metrics and among university or adult learners [1, 71, 85]. Abu Deeb and Hickey (2021) calculated the number of runs students performed before giving up on a bug, and Jemmali et al. (2022) and Lee et al. (2011) calculated the number of levels participants attempted in their

debugging game [71, 85]. These studies paint a mixed picture of how interventions can influence debugging persistence. In a naturalistic coding environment, Abu Deeb and Hickey (2021) found that their metacognitive scaffolding tool, Spinoza, *decreased* the amount of time students spent before they gave up on the bug [1]. Authors attributed this finding to the unintuitive design of the system interface, causing some students to refresh or give up. In a game setting, Lee et al. (2011) showed that when participants were provided with personified compiler feedback, they completed more levels [85]. Jemmali et al. (2022) also found that students attempted more levels in a debugging game when given personalized debugging exercises [71]. However, this difference did not hold when the personalization was removed. These findings tentatively suggest that personification and personalization can improve persistence in game settings, but further research is needed to understand how to sustain this improvement when students debug their own code in naturalistic settings.

Other. Sixteen papers in our sample also measured constructs not categorized in our coding scheme. Other outcome measures included self-reported ratings of how much they enjoyed programming or robotics [71, 91, 105, 131, 149], cognitive measures such as mental-load [149], productivity [80], and overall course exam results or pass-rates [123].

5 Discussion

This research provides an overview of debugging interventions by examining 43 papers from 2010 to 2022. Our sample shows an increase in interventions within the past few years and a diverse array of designs, ranging from games, to workshops, to unplugged activities. While the majority of interventions are designed with learning mechanisms in mind, we also note that papers did not consistently list or consider their pedagogical approach. We also note a scarcity of replication studies in our sample, highlighting the importance of future debugging education research to validate and extend existing findings. Our assessment of efficacy reveals promising results, confirming that interventions can improve debugging accuracy and learning. However, interventions showed less success in helping students adopt systematic debugging strategies. A handful of studies demonstrate that interventions can support non-cognitive constructs related to debugging, such as self-efficacy and mindset, particularly among K-12 learners. While these results are limited to a small sample of papers, they provide encouraging motivation for the field to explore additional non-cognitive interventions, particularly among university and adult learners. We discuss our findings in further detail to highlight implications for research and instruction.

5.1 Implications for Research

Our main coding scheme categorized the interventions in our sample by modality, pedagogical approach, debugging step targeted, non-cognitive skill, and efficacy. The findings from our primary analysis reveal gaps in the literature and promising areas for future research. In particular, we highlight less targeted cognitive and non-cognitive debugging skills and opportunities to improve the transfer and adoption of these skills.

Additional Interventions to Support Code Comprehension and Reflection. Our results classifying the debugging steps revealed that interventions designed to support Steps 1 and 5 of the debugging process were under-researched (Table 5). Comparatively fewer papers in our sample emphasized the first step of constructing a mental model of the code—a trend that aligns with students' tendency to overlook this step as well [53, 140]. It is worth noting that the limited focus on Step 1 may reflect its primary relevance to debugging *foreign* code rather than one's own code. While fixing one's own code is more common in introductory courses, debugging foreign code is still important in various contexts, including working with APIs and libraries, collaborating with others [84], and interpreting

artificial intelligence (AI)-generated code [38]. This highlights a need to teach students code comprehension and the importance of *first* understanding the code before actively searching for errors. We note that a rich literature in CS education has focused on designing interventions and tools to improve program comprehension (e.g., [70, 110, 147]), which can provide helpful starting points for future debugging interventions. Additionally, we found that comparatively fewer interventions were designed to support meaningful reflection and documentation after debugging (Step 5). Encouraging reflection can be a difficult design task. In their work investigating self-regulation in novices, Loksa et al. (2020) highlight that *in situ* reflection can interrupt the programming process, yet it is difficult to recall one's thoughts and feelings during the task afterward [93]. To navigate this delicate balance, future work can explore methods of capturing pivotal debugging moments *in situ* while crafting questions to prompt meaningful reflection about these moments afterward. In our results, we also found that while many interventions *passively* supported students in testing their code (Steps 2 and 4) by providing pre-written test cases, students did not need to actively engage in testing. Future interventions can explore more scaffolded methods to support code testing for debugging, such as in the case of Pechorina et al. (2023), where students must first solve the test case before using it [115].

Expanding Measurements for Non-Cognitive Constructs. Our findings also highlight an opportunity for more interventions supporting the non-cognitive aspects of debugging—self-efficacy, mindset, affect, and persistence—particularly among university and adult learners. Less than a fourth of the papers we reviewed explicitly discussed how their interventions might target these dimensions, and only a fraction conducted evaluations of non-cognitive constructs. Statistically assessed interventions targeting affect and persistence occasionally demonstrated mixed and even detrimental outcomes.

Our visualization of assessment methods (Figure 5) reveals opportunities to explore more nuanced measures for non-cognitive constructs. For example, Figure 5 shows that affect was primarily assessed *after* the debugging task using interviews and surveys, but observational measures of student emotions *while* debugging may be more informative of emotion regulation. Additionally, the debugging mindset was solely assessed through interviews and written reflections. While these provide rich insights, developing a validated survey for the debugging mindset may encourage more researchers to target this as an intervention outcome. In general, establishing more expansive measures to assess how interventions influence non-cognitive constructs is a promising area of future research. For example, to better understand how an intervention is helping students regulate emotions, researchers can measure *in situ* self-reports of emotions and electrodermal activity. These have been explored for general programming tasks but not specifically for debugging [55, 56]. Additionally, to effectively study the impact of interventions on persistence, it may be valuable to examine more long-term process measures, such as whether students demonstrate sustained persistence when tackling bugs in their own code. Lastly, few studies used validated surveys for measuring these non-cognitive constructs. The field may benefit from developing and consolidating validated measures of these outcomes, such as [127] for debugging self-efficacy and anxiety.

Targeting Adoption and Transfer Learning. Our evaluation of intervention efficacy revealed that students did not consistently adopt systematic strategies taught to them. The authors of these studies posit several potential explanations [24, 80, 141, 144]: (1) students may lack confidence in executing debugging strategies, (2) the use of strategies requires substantial self-regulation and domain knowledge, which students may not have, (3) students may perceive that the strategies are slowing them down, and (4) students may lack engagement with the debugging task. The first two explanations suggest that additional cognitive and non-cognitive skills are required to adopt a more systematic process. Future research can explore teaching debugging instruction

alongside other skills, such as self-regulation or systematic reasoning. Additionally, the third and fourth interpretations suggest that motivation plays a large role in strategy adoption. There is an opportunity to research effective approaches to reinforce the importance of adopting a systematic approach to debugging. Whalley et al. (2021) suggest that introducing more difficult bugs and prompting students to reflect on their process may increase a student's value of a systematic debugging approach [144].

Lastly, our results show evidence that students learned how to debug more accurately following an intervention. However, this was not always the case in terms of debugging efficiency. In four studies, students debugged more efficiently *while* using the tool, but these improvements did not transfer when debugging without it. This insight suggests that future interventions can explore methods to phase out support as learners progress, facilitating their development of independent debugging skills.

*Exploring **Large Language Models (LLMs)** to Scaffold Debugging Instruction.* Lastly, an important area for future research is understanding how LLMs and AI-assisted programming tools may change debugging practices and instruction. The current review provides a "snapshot" of debugging interventions prior to the influx of LLM technologies. As these technologies evolve, studies should examine their effects on students' debugging skills, strategy adoption, and the overall approach to teaching debugging in CS education. With the availability of LLMs, researchers anticipate a potential shift in the traditional coding paradigm to emphasize *debugging* generated code over code *generation* [38]. This underscores the importance of debugging instruction and introduces possibilities to leverage LLMs in scaffolding such instruction. Prior research has demonstrated that LLMs can execute each step of the debugging process, including explaining code (Step 1) [98], rewriting error messages [87] (Step 2), diagnosing bugs in the code (Step 3) [50], and fixing them (Step 4) [50]. A promising area of future research is to explore how LLMs can scaffold students' skills throughout the debugging process when correcting their own code. Another potential application is for LLMs to generate debugging exercises derived from students' code or errors, offering a platform for iterative and personalized debugging practice.

5.2 Implications for Instruction

In recognizing the challenges instructors may encounter when teaching students how to debug, another goal of this literature review is to highlight effective approaches for debugging instruction. While we believe that teachers offer the most personalized form of instruction, we synthesized a range of ideas from the literature to provide additional pedagogical approaches for instructors to explore. Section 4.2 summarizes pedagogical approaches for debugging, accompanied by explanations and examples. Table 5 offers an overview of methods to target the various cognitive and non-cognitive skills required for debugging. In addition to teaching a systematic process of debugging to students, instructors can reference this table to explore methods to support each step.

Overall, our findings indicate that pedagogical approaches work best when used in combination. For example, direct instruction was frequently coupled with deliberate practice of the taught strategies and processes, showing positive results for both K-12 and adult learners [104, 105]. Observational learning through modeling effective strategies can be beneficial for students, especially when paired with metacognitive prompts to highlight why certain strategies are being used [36, 80]. With younger students using block-programming languages, embodying the code combined with translating the code to natural language (scaffolding) was most effective for learning [4]. Instructors should explore different ways of combining pedagogical approaches to boost student participation and learning.

Paper insights also suggest that encouraging students to *adopt* systematic debugging strategies and processes can be challenging. To execute these strategies, students must have adequate self-regulation, domain knowledge, and technical skills. Since many students do not enter introductory courses with these skills, debugging instruction should be offered *consistently* throughout the course, not just at the beginning. As students master basic debugging strategies and encounter more complex bugs, instructors can gradually introduce more sophisticated and advanced debugging approaches and examples. Additionally, since there are "good" and "bad" ways of using debugging strategies [108], it is important to teach the reasoning behind effective strategy usage. For instance, explicitly teaching students *where* to put print statements for effective tracing or *how* to generate targeted hypotheses about the bug's location.

5.3 Limitations

Several factors in the search, selection, data extraction, and interpretation process could influence the validity of the findings presented. To mitigate the risk of validity threats, we followed recommendations proposed within the PRISMA framework [113] and the set of guidelines presented by Ampatzoglou et al. (2019) for software engineering reviews [9].

During the search process, we iteratively tested our search terms for completeness and appropriateness, to strengthen the external validity of our findings. Still, our search terms may have failed to capture relevant papers. Research within computing education related to debugging instruction may not have listed the specific term "debug" in the abstract, leading us to exclude it from the sample. Additionally, because "learning" was a specific focus of our review and a related keyword, we may have unintentionally excluded papers solely focused on the non-cognitive aspects of debugging, leading to a bias in our sample and conclusions. In our selection process, we conducted multiple rounds of screening to ensure that papers presented high-quality research and targeted debugging *learning*. To specifically screen for papers intended to *teach* debugging, we identified explicit "learning intent" statements within the papers. However, we acknowledge that excluded interventions may have had pedagogical benefits, even if the authors did not explicitly state this to be a focus of their design or results. During the data extraction process, we conducted multiple rounds of discussion to create our coding scheme and achieve a high level of consensus and IRR. Although we took a top-down and bottom-up approach to generate our coding scheme, we acknowledge that our categories may not be comprehensive. For example, we identified self-efficacy, mindset, affect, and persistence as the primary non-cognitive constructs related to debugging in our sample, but we acknowledge that there may be other important non-cognitive constructs for debugging.

Lastly, to address our fourth research question regarding intervention efficacy, we decided not to conduct a meta-analysis, due to the small sample of papers that empirically tested their intervention, which is needed to determine effect size. Instead, we coded the results of significance testing for papers that assessed their interventions quantitatively and summarized the main findings from qualitative results. While we draw important insights from these results, we acknowledge that these methods cannot truly estimate the overall treatment effect of interventions in our sample. Future reviews should conduct more in-depth studies with a larger sample size to quantitatively estimate the efficacy of debugging interventions across different categories of intervention design and assessed outcome. Furthermore, we acknowledge the potential influence of publication bias on our findings, particularly regarding the reported success of debugging interventions in terms of accuracy and efficiency. It is possible that studies showing positive results are more likely to be published, while those with null or negative findings may remain unpublished. This bias could lead to an overestimation of the overall effectiveness of debugging interventions. We also acknowledge that conclusions about efficacy were often derived from a small sample of papers, since we categorized them by measured construct, and only a few papers assessed certain constructs.

Thus, although our findings highlight notable gaps in the literature, efficacy results should be interpreted with caution due to the small sample size.

6 Conclusion

Debugging errors in code is a major hurdle for beginner programmers. In CS courses, students often do not receive adequate instruction on debugging, and instructors report uncertainty about teaching these skills. We presented a systematic review of interventions from 2010 to 2022 designed to teach debugging. To support instructors, we summarized foundational pedagogical approaches and methods to target the various cognitive and non-cognitive skills associated with debugging. To support researchers in designing future interventions, we summarized methods and metrics to evaluate debugging and highlight opportunities for future work. Our findings reveal successful results, confirming that interventions can improve debugging accuracy and learning. Still, there are opportunities for interventions to support the non-cognitive skills associated with debugging, such as emotion regulation and adopting systematic strategies. Future research should also explore how emerging technologies, including large language models, may reshape debugging practices and instruction methods.

References

[1] Fatima Abu Deeb and Timothy Hickey. 2021. Reflective debugging in Spinoza V3.0. In *Proceedings of the Australasian Computing Education Conference (ACE '21)*. ACM, New York, NY, 125–130. DOI: https://doi.org/10.1145/3441636.3442313

[2] Marzieh Ahmadzadeh, Dave Elliman, and Colin Higgins. 2005. An analysis of patterns of debugging among novice computer science students. *SIGCSE Bull.* 37, 3 (Jun. 2005), 84–88. DOI: https://doi.org/10.1145/1151954.1067472

[3] Umair Z. Ahmed, Nisheeth Srivastava, Renuka Sindhgatta, and Amey Karkare. 2020. Characterizing the pedagogical benefits of adaptive feedback for compilation errors by novice programmers. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET '20)*. ACM, New York, NY, 139–150. DOI: https://doi.org/10.1145/3377814.3381703

[4] Junghyun Ahn, Woonhee Sung, and John B. Black. 2022. Unplugged debugging activities for developing young learners' debugging skills, 421–437. DOI: https://doi.org/10.1080/02568543.2021.1981503

[5] Vincent Aleven, Bruce McLaren, Ido Roll, and Kenneth Koedinger. 2006. Toward meta-cognitive tutoring: A model of help seeking with a cognitive tutor. *Int. J. Artif. Intell. Educ.* 16, 2 (Jan. 2006), 101–128.

[6] Amal Alhadabi and Aryn C. Karpinski. 2020. Grit, self-efficacy, achievement orientation goals, and academic performance in University students. *Int. J. Adolesc. Youth* 25, 1 (Dec. 2020), 519–535. DOI: https://doi.org/10.1080/02673843.2019.1679202

[7] Basma S. Alqadi and Jonathan I. Maletic. 2017. An empirical study of debugging patterns among novices programmers. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education (SIGCSE '17)*. ACM, New York, NY, 15–20. DOI: https://doi.org/10.1145/3017680.3017761

[8] Malek Alrashidi, Michael Gardner, and Vic Callaghan. 2017. Evaluating the use of pedagogical virtual machine with augmented reality to support learning embedded computing activity. In *Proceedings of the 9th International Conference on Computer and Automation Engineering (ICCAE '17)*. ACM, New York, NY, 44–50. DOI: https://doi.org/10.1145/3057039.3057088

[9] Apostolos Ampatzoglou, Stamatia Bibi, Paris Avgeriou, Marijn Verbeek, and Alexander Chatzigeorgiou. 2019. Identifying, categorizing and mitigating threats to validity in software engineering secondary studies. *Inf. Softw. Technol.* 106 (Feb. 2019), 201–230. DOI: https://doi.org/10.1016/j.infsof.2018.10.006

[10] Pasquale Ardimento, Mario Luca Bernardi, Marta Cimitile, and Giuseppe De Ruvo. 2019. Reusing bugged source code to support novice programmers in debugging tasks. *ACM Trans. Comput. Educ.* 20, 1 (Nov. 2019), 1–24. DOI: https://doi.org/10.1145/3355616

[11] Ryan S. J. d. Baker, Sidney K. D'Mello, Ma Mercedes T. Rodrigo, and Arthur C. Graesser. 2010. Better to be frustrated than bored: The incidence, persistence, and impact of learners' cognitive–affective states during interactions with three different computer-based learning environments. *Int. J. Hum. Comput. Stud.* 68, 4 (Apr. 2010), 223–241. DOI: https://doi.org/10.1016/j.ijhcs.2009.12.003

[12] Albert Bandura. 1977. Self-efficacy: Toward a unifying theory of behavioral change. *Psychol. Rev.* 84, 2 (Mar. 1977), 191–215. DOI: https://doi.org/10.1037//0033-295x.84.2.191

[13] Albert Bandura. 1982. Self-efficacy mechanism in human agency. *Am. Psychol.* 37, 2 (1982), 122–147. DOI: https://doi.org/10.1037//0003-066x.37.2.122

[14] Albert Bandura. 1986. *Social Foundations of Thought and Action: A Social Cognitive Theory*. Prentice-Hall, Englewood Cliffs, NJ.

[15] Lawrence W. Barsalou. 2008. Grounded cognition. *Annu. Rev. Psychol.* 59, 1 (Jan. 2008), 617–645. DOI: https://doi.org/10.1146/annurev.psych.59.103006.093639

[16] Joseph E. Beck and Yue Gong. 2013. Wheel-spinning: Students who fail to master a skill. In *Artificial Intelligence in Education*. Springer, Berlin, 431–440. DOI: https://doi.org/10.1007/978-3-642-39112-5_44

[17] B. A. Becker, P. Denny, R. Pettit, D. Bouchard, Dennis J. Bouvier, Brian Harrington, Amir Kamil, Amey Karkare, Chris McDonald, Peter-Michael Osera, Janice L. Pearce, and James Prather. 2019. Compiler error messages considered unhelpful: The landscape of text-based programming error message research. In *Proceedings of the Working Group Reports on Innovation and Technology in Computer Science Education*, 177–210.

[18] Brett A. Becker, Kyle Goslin, and Graham Glanville. 2018. The effects of enhanced compiler error messages on a syntax error debugging test. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education (SIGCSE '18)*. ACM, New York, NY, 640–645. DOI: https://doi.org/10.1145/3159450.3159461

[19] Roman Bednarik, Carsten Schulte, Lea Budde, Birte Heinemann, and Hana Vrzakova. 2018. Eye-movement modeling examples in source code comprehension: A classroom study. In *Proceedings of the 18th Koli Calling International Conference on Computing Education Research (Koli Calling '18, Article 2)*. ACM, New York, NY, 1–8. DOI: https://doi.org/10.1145/3279720.3279722

[20] Susan Bergin, Ronan Reilly, and Desmond Traynor. 2005. Examining the role of self-regulated learning on introductory programming performance. In *Proceedings of the 1st International Workshop on Computing Education Research (ICER '05)*. ACM, New York, NY, 81–86. DOI: https://doi.org/10.1145/1089786.1089794

[21] Katerine Bielaczyc, Peter L. Pirolli, and Ann L. Brown. 1995. Training in self-explanation and self-regulation strategies: Investigating the effects of knowledge acquisition activities on problem solving. *Cogn. Instr.* 13, 2 (1995), 221–252.

[22] Laura Bofferding, Sezai Kocabas, Mahtob Aqazade, Ana-Maria Haiduc, and Lizhen Chen. 2022. The effect of play and worked examples on first and third graders' creating and debugging of programming algorithms. In *Proceedings of the Computational Thinking in PreK-5: Empirical Evidence for Integration and Future Directions*. ACM, New York, NY, 19–29. DOI: https://doi.org/10.1145/3507951.3519284

[23] Nigel Bosch and Sidney D'Mello. 2017. The affective experience of novice computer programmers. *Int. J. Artif. Intell. Educ.* 27, 1 (Mar. 2017), 181–206. DOI: https://doi.org/10.1007/s40593-015-0069-5

[24] Axel Böttcher, Veronika Thurner, Kathrin Schlierkamp, and Daniela Zehetmeier. 2016. Debugging students' debugging process. In *Proceedings of the 2016 IEEE Frontiers in Education Conference (FIE '16)*, 1–7. DOI: https://doi.org/10.1109/FIE.2016.7757447

[25] Karen Brennan and Mitchel Resnick. 2012. New frameworks for studying and assessing the development of computational thinking. In *Proceedings of the 2012 Annual Meeting of the American Educational Research Association*, Vol. 1, 25.

[26] Randal E. Bryant and David R. O'Hallaron. 2001. Introducing computer systems from a programmer's perspective. In *Proceedings of the 32nd SIGCSE Technical Symposium on Computer Science Education (SIGCSE '01)*. ACM, New York, NY, 90–94. DOI: https://doi.org/10.1145/364447.364549

[27] Sallyann Bryant, Pablo Romero, and Benedict du Boulay. 2008. Pair programming and the mysterious role of the navigator. *Int. J. Hum. Comput. Stud.* 66, 7 (Jul. 2008), 519–529. DOI: https://doi.org/10.1016/j.ijhcs.2007.03.005

[28] Elizabeth Carter. 2015. Its debug: Practical results. *J. Comput. Sci. Coll.* 30, 3 (Jan. 2015), 9–15.

[29] Mccoy Sharon Carver and Sally Clarke Risinger. 1987. Improving children's debugging skills. In *Empirical Studies of Programmers: Second Workshop*. Ablex Publishing Corp., 147–171.

[30] Xingliang Chen, Antonija Mitrovic, and Moffat Mathews. 2020. Learning from Worked Examples, Erroneous Examples, and Problem Solving: Toward Adaptive Selection of Learning Activities. *IEEE Trans. Learn. Technol.* 13, 1 (2020), 135–149. DOI: https://doi.org/10.1109/TLT.2019.2896080

[31] Ryan Chmiel and Michael C. Loui. 2004. Debugging: From novice to expert. In *Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education (SIGCSE '04)*. ACM, New York, NY, 17–21. DOI: https://doi.org/10.1145/971300.971310

[32] Cheng-Yu Chung and I.-Han Hsiao. 2020. Computational thinking in augmented reality: An investigation of collaborative debugging practices. In *Proceedings of the 2020 6th International Conference of the Immersive Learning Research Network (iLRN '20)*. ieeexplore.ieee.org, 54–61. DOI: https://doi.org/10.23919/iLRN47897.2020.9155152

[33] James Cross, Dean Hendrix, Larry Barowski, and David Umphress. 2014. Dynamic program visualizations: An experience report. In *Proceedings of the 45th ACM Technical Symposium on Computer Science Education (SIGCSE '14)*. ACM, New York, NY, 609–614. DOI: https://doi.org/10.1145/2538862.2538958

[34] Maggie Dahn, David Deliema, and Noel Enyedy. 2020. Art as a point of departure for understanding student experience in learning to code. *Teach. Coll. Rec.* 122, 8 (Aug. 2020), 1–42. DOI: https://doi.org/10.1177/016146812012200802

[35] Sean Deitz and Ugo Buy. 2016. From video games to debugging code. In *Proceedings of the 5th International Workshop on Games and Software Engineering (GAS '16)*. ACM, New York, NY, 37–41. DOI: https://doi.org/10.1145/2896958.2896964

[36] David DeLiema, Maggie Dahn, Virginia J. Flood, and Francis F. Steen. 2019. Debugging as a context for fostering reflection on critical thinking and emotion. In *Deeper Learning, Dialogic Learning, and Critical Thinking*, 209–228. DOI: https://doi.org/10.4324/9780429323058-13

[37] Paul Denny, James Prather, and Brett A. Becker. 2020. Error message readability and novice debugging performance. In *Proceedings of the 2020 ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE '20)*. ACM, New York, NY, 480–486. DOI: https://doi.org/10.1145/3341525.3387384

[38] Paul Denny, James Prather, Brett A Becker, James Finnie-Ansley, Arto Hellas, Juho Leinonen, Andrew Luxton-Reilly, Brent N Reeves, Eddie Antonio Santos, and Sami Sarsa. 2024. Computing education in the era of generative AI. *Commun. ACM* 67, 2 (Feb 2024), 56–67. DOI: https://doi.org/10.1145/3624720

[39] Sidney D'Mello and Art Graesser. 2012. Dynamics of affective states during complex learning. *Learn. Instr.* 22, 2 (Apr. 2012), 145–157. DOI: https://doi.org/10.1016/j.learninstruc.2011.10.001

[40] M. Ducassé and A.-M. Emde. 1988. A review of automated debugging systems: knowledge, strategies and techniques. In *Proceedings of the 10th international conference on Software engineering (ICSE '88)*. IEEE Computer Society Press, Washington, DC, 162–171.

[41] Angela L. Duckworth, Christopher Peterson, Michael D. Matthews, and Dennis R. Kelly. 2007. Grit: Perseverance and passion for long-term goals. *J. Pers. Soc. Psychol.* 92, 6 (Jun. 2007), 1087–1101. DOI: https://doi.org/10.1037/0022-3514.92.6.1087

[42] Henry Duwe, Diane T. Rover, Phillip H. Jones, Nicholas D. Fila, and Mani Mina. 2022. Defining and supporting a debugging mindset in computer engineering courses. In *Proceedings of the 2022 IEEE Frontiers in Education Conference (FIE '22)*, 1–9. DOI: https://doi.org/10.1109/FIE56618.2022.9962605

[43] C. S. Dweck. 1990. Self-theories and goals: Their role in motivation, personality, and development. *Nebr. Symp. Motiv.* 38 (1990), 199–235.

[44] Carol S. Dweck. 2006. *Mindset: The New Psychology of Success*. Random House Publishing Group.

[45] Carol S. Dweck, Gregory M. Walton, and Geoffrey L. Cohen. 2014. *Academic Tenacity: Mindsets and Skills that Promote Long-Term Learning*. Bill & Melinda Gates Foundation.

[46] Bob Edmison and Stephen H. Edwards. 2020. Turn up the heat! using heat maps to visualize suspicious code to help students successfully complete programming problems faster. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET '20)*. ACM, New York, NY, 34–44. DOI: https://doi.org/10.1145/3377814.3381707

[47] Kiran L. N. Eranki and Kannan M. Moudgalya. 2016. Program slicing technique: A novel approach to improve programming skills in novice learners. In *Proceedings of the 17th Annual Conference on Information Technology Education (SIGITE '16)*. ACM, New York, NY, 160–165. DOI: https://doi.org/10.1145/2978192.2978215

[48] K. Anders Ericsson. 2008. Deliberate practice and acquisition of expert performance: A general overview. *Acad. Emerg. Med.* 15, 11 (Nov. 2008), 988–994. DOI: https://doi.org/10.1111/j.1553-2712.2008.00227.x

[49] Richard Falk and Samuel S. Kim. 2019. *The War System: An Interdisciplinary Approach*. Routledge.

[50] Zhiyu Fan, Xiang Gao, Martin Mirchev, Abhik Roychoudhury, and Shin Hwei Tan. 2023. Automated repair of programs from large language models. In *Proceedings of the 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE '23)*. IEEE, 1469–1481. DOI: https://doi.org/10.1109/ICSE48619.2023.00128

[51] Joel Fenwick and Peter Sutton. 2012. Using quicksand to improve debugging practice in post-novice level students. In *Proceedings of the Fourteenth Australasian Computing Education Conference* 123 (2012), 141–146.

[52] Deborah A. Fields, Yasmin B. Kafai, Luis Morales-Navarro, and Justice T. Walker. 2021. Debugging by design: A constructionist approach to high school students' crafting and coding of electronic textiles as failure artefacts. *Br. J. Educ. Technol.* 52, 3 (May 2021), 1078–1092. DOI: https://doi.org/10.1111/bjet.13079

[53] Sue Fitzgerald, Gary Lewandowski, Renée McCauley, Laurie Murphy, Beth Simon, Lynda Thomas, and Carol Zander. 2008. Debugging: Finding, fixing and flailing, a multi-institutional study of novice debuggers. *Comput. Sci. Educ.* 18, 2 (Jun. 2008), 93–116. DOI: https://doi.org/10.1080/08993400802114508

[54] Rita Garcia, Chieh-Ju Liao, and Ariane Pearce. 2022. Read the debug manual: A debugging manual for CS1 students. In *Proceedings of the 2022 IEEE Frontiers in Education Conference (FIE '22)*. ieeexplore.ieee.org, 1–7.

[55] Daniela Girardi, Nicole Novielli, Davide Fucci, and Filippo Lanubile. 2020. Recognizing developers' emotions while programming. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering (ICSE '20)*. ACM, New York, NY, 666–677. DOI: https://doi.org/10.1145/3377811.3380374

[56] Jamie Gorson, Kathryn Cunningham, Marcelo Worsley, and Eleanor O'Rourke. 2022. Using electrodermal activity measurements to understand student emotions while programming. In *Proceedings of the 2022 ACM Conference on International Computing Education Research (ICER '22)*, Vol. 1, ACM, New York, NY, 105–119. DOI: https://doi.org/10.1145/3501385.3543981

[57] Jamie Gorson and Eleanor O'Rourke. 2020. Why do CS1 students think they're bad at programming? Investigating self-efficacy and self-assessments at three universities. In *Proceedings of the 2020 ACM Conference on International Computing Education Research (ICER '20)*. ACM, New York, NY, 170–181. DOI: https://doi.org/10.1145/3372782.3406273

[58] Luisa Greifenstein, Florian Obermueller, Ewald Wasmeier, Ute Heuer, and Gordon Fraser. 2021. Effects of hints on debugging scratch programs: An empirical study with primary school teachers in training. In *Proceedings of the 16th Workshop in Primary and Secondary Computing Education (WiPSCE '21, Article 3)*. ACM, New York, NY, 1–10. DOI: https://doi.org/10.1145/3481312.3481344

[59] Paul Gross and Kris Powers. 2005. Evaluating assessments of novice programming environments. In *Proceedings of the 1st International Workshop on Computing Education Research (ICER '05)*. ACM, New York, NY, 99–110. DOI: https://doi.org/10.1145/1089786.1089796

[60] L. Gugerty and G. Olson. 1986. Debugging by skilled and novice programmers. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '86)*. ACM, New York, NY, 171–174. DOI: https://doi.org/10.1145/22627.22367

[61] Philip J. Guo. 2013. Online python tutor: Embeddable web-based program visualization for CS education. In *Proceedings of the 44th ACM technical symposium on Computer science education (SIGCSE '13)*. ACM, New York, NY, 579–584. DOI: https://doi.org/10.1145/2445196.2445368

[62] B. Hailpern and P. Santhanam. 2002. Software debugging, testing, and verification. *IBM Syst. J.* 41, 1 (2002), 4–12. DOI: https://doi.org/10.1147/sj.411.0004

[63] Brian Hanks, Sue Fitzgerald, Renée McCauley, Laurie Murphy, and Carol Zander. 2011. Pair programming in education: A literature review. *Comput. Sci. Educ.* 21, 2 (Jun. 2011), 135–173. DOI: https://doi.org/10.1080/08993408.2011.579808

[64] Björn Hartmann, Daniel MacDougall, Joel Brandt, and Scott R. Klemmer. 2010. What would other programmers do: Suggesting solutions to error messages. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '10)*. ACM, New York, NY, 1019–1028. DOI: https://doi.org/10.1145/1753326.1753478

[65] Sami Heikkinen, Mohammed Saqr, Jonna Malmberg, and Matti Tedre. 2023. Supporting self-regulated learning with learning analytics interventions – A systematic literature review. *Educ. Inf. Tech.* 28, 3 (March 2023), 3059–3088. DOI: https://doi.org/10.1007/s10639-022-11281-4

[66] R. L. Heilman and G. P. Ashby. 1971. Re-evaluation of debugging in the computer science curriculum. *SIGCSE Bull.* 3, 4 (Dec. 1971), 15–18. DOI: https://doi.org/10.1145/382214.382215

[67] Juha Helminen and Lauri Malmi. 2010. Jype - A program visualization and programming exercise tool for Python. In *Proceedings of the 5th International Symposium on Software Visualization (SOFTVIS '10)*. ACM, New York, NY, 153–162. DOI: https://doi.org/10.1145/1879211.1879234

[68] Matthew Hertz and Maria Jump. 2013. Trace-based teaching in early programming courses. In *Proceedings of the 44th ACM Technical Symposium on Computer Science Education (SIGCSE '13)*. ACM, New York, NY, 561–566. DOI: https://doi.org/10.1145/2445196.2445364

[69] Mark A. Holliday and David Luginbuhl. 2004. CS1 assessment using memory diagrams. In *Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education (SIGCSE '04)*. ACM, New York, NY, 200–204. DOI: https://doi.org/10.1145/971300.971373

[70] Cruz Izu, Carsten Schulte, Ashish Aggarwal, and Renske Weeda. 2019. Fostering program comprehension in novice programmers - Learning activities and learning trajectories. In *Innovation and Technology in Computer Science Education (ITiCSE '19)*, 27–52. DOI: https://doi.org/10.1145/3344429.3372501

[71] Chaima Jemmali, Magy Seif El-Nasr, and Seth Cooper. 2022. The effects of adaptive procedural levels on engagement and performance in an educational programming game. In *Proceedings of the 17th International Conference on the Foundations of Digital Games (FDG '22)*, 1–12. DOI: https://doi.org/10.1145/3555858.3555892

[72] Mina C. Johnson-Glenberg, David A. Birchfield, Lisa Tolentino, and Tatyana Koziupa. 2014. Collaborative embodied learning in mixed reality motion-capture environments: Two science studies. *J. Educ. Psychol.* 106, 1 (Feb. 2014), 86–104. DOI: https://doi.org/10.1037/a0034008

[73] David H. Jonassen and Woei Hung. 2006. Learning to troubleshoot: A new theory-based design architecture. *Educ. Psychol. Rev.* 18, 1 (Mar. 2006), 77–114. DOI: https://doi.org/10.1007/s10648-006-9001-8

[74] Irvin R. Katz and John R. Anderson. 1987. Debugging: An analysis of bug-location strategies. *Hum.–Comput. Interact.* 3, 4 (Dec. 1987), 351–399. https://doi.org/10.1207/s15327051hci0304_2

[75] Claudius M. Kessler and John R. Anderson. 1986. A model of novice debugging in LISP. In *Papers Presented at the First Workshop on Empirical Studies of Programmers on Empirical Studies of Programmer*. Ablex Publishing Corp., 198–212.

[76] Päivi Kinnunen and Lauri Malmi. 2006. Why students drop out CS1 course? In *Proceedings of the 2nd International Workshop on Computing Education Research (ICER '06)*. ACM, New York, NY, 97–108. DOI: https://doi.org/10.1145/1151588.1151604

[77] Päivi Kinnunen and Beth Simon. 2012. My program is ok – Am I? Computing Freshmen's experiences of doing programming assignments. *Comput. Sci. Educ.* 22, 1 (Mar. 2012), 1–28. DOI: https://doi.org/10.1080/08993408.2012.655091

[78] David Klahr and Sharon Mccoy Carver. 1988. Cognitive objectives in a LOGO debugging curriculum: Instruction, learning, and transfer. *Cogn. Psychol.* 20, 3 (Jul. 1988), 362–404. DOI: https://doi.org/10.1016/0010-0285(88)90004-7

[79] Donald E. Knuth. 1989. The errors of tex. *Softw. Pract. Exp.* 19, 7 (Jul. 1989), 607–685. DOI: https://doi.org/10.1002/spe.4380190702

[80] 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 (SIGCSE '19)*. ACM, New York, NY, 469–475. DOI: https://doi.org/10.1145/3287324.3287371

[81] Michael Kölling. 2008. Using BlueJ to introduce programming. In *Reflections on the Teaching of Programming: Methods and Implementations*. Jens Bennedsen, Michael E. Caspersen, and Michael Kölling (Eds.), Springer, Berlin, 98–115. DOI: https://doi.org/10.1007/978-3-540-77934-6_9

[82] Donna Kotsopoulos, Lisa Floyd, Steven Khan, Immaculate Kizito Namukasa, Sowmya Somanath, Jessica Weber, and Chris Yiu. 2017. A pedagogical framework for computational thinking. *Digital Experiences in Mathematics Education* 3, 2 (Aug. 2017), 154–171. https://doi.org/10.1007/s40751-017-0031-2

[83] Essi Lahtinen, Kirsti Ala-Mutka, and Hannu-Matti Järvinen. 2005. A study of the difficulties of novice programmers. *SIGCSE Bull.* 37, 3 (Jun. 2005), 14–18. DOI: https://doi.org/10.1145/1151954.1067453

[84] Thomas D. Latoza, Gina Venolia, and R. Deline. 2006. Maintaining mental models: A study of developer work habits. *Int. Conf. Softw. Eng.* (May 2006). DOI: https://doi.org/10.1145/1134285.1134355

[85] Michael J. Lee and Amy J. Ko. 2011. Personifying programming tool feedback improves novice programmers' learning. In *Proceedings of the 7th International Workshop on Computing Education Research (ICER '11)*. ACM, New York, NY, 109–116. DOI: https://doi.org/10.1145/2016911.2016934

[86] V. C. S. Lee, Y. T. Yu, C. M. Tang, T. L. Wong, and C. K. Poon. 2018. ViDA: A virtual debugging advisor for supporting learning in computer programming courses. *J. Comput. Assist. Learn.* 34, 3 (Jun. 2018), 243–258. DOI: https://doi.org/10.1111/jcal.12238

[87] J. Leinonen, A. Hellas, S. Sarsa, B. Reeves, P. Denny, J. Prather, and B. Becker. 2023. Using large language models to enhance programming error messages. *Proceedings of the 54th ACM Technical Symposium on Computer Science Education* 1 (2023), 563–569.

[88] Chen Li, Emily Chan, Paul Denny, Andrew Luxton-Reilly, and Ewan Tempero. 2019. Towards a framework for teaching debugging. In *Proceedings of the 21st Australasian Computing Education Conference (ACE '19)*. ACM, New York, NY, 79–86. DOI: https://doi.org/10.1145/3286960.3286970

[89] Yu-Tzu Lin, Cheng-Chih Wu, Ting-Yun Hou, Yu-Chih Lin, Fang-Ying Yang, and Chia-Hu Chang. 2016. Tracking students' cognitive processes during program debugging—An eye-movement approach. *IEEE Trans. Educ.* 59, 3 (Aug. 2016), 175–186. DOI: https://doi.org/10.1109/TE.2015.2487341

[90] A. Lishinski, A. Yadav, and R. Enbody. 2017. Students' emotional reactions to programming projects in introduction to programming: Measurement approach and influence on learning outcomes. In *Proceedings of the 2017 ACM Conference on International Computing Education Research* (2017).

[91] Zhongxiu Liu, Rui Zhi, Andrew Hicks, and Tiffany Barnes. 2017. Understanding problem solving behavior of 6–8 graders in a debugging game. *Comput. Sci. Educ.* 27, 1 (Jan. 2017), 1–29. DOI: https://doi.org/10.1080/08993408.2017.1308651

[92] 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 (ICER '16)*. ACM, New York, NY, 83–91. DOI: https://doi.org/10.1145/2960310.2960334

[93] Dastyni Loksa, Benjamin Xie, Harrison Kwik, and Amy J. Ko. 2020. Investigating novices' in situ reflections on their programming process. In *Proceedings of the 51st ACM Technical Symposium on Computer Science Education (SIGCSE '20)*. ACM, New York, NY, 149–155. DOI: https://doi.org/10.1145/3328778.3366846

[94] R. Luckin, K. R. Koedinger, and J. Greer. 2007. *Artificial Intelligence in Education: Building Technology Rich Learning Contexts that Work*. IOS Press.

[95] Andrew Luxton-Reilly, Emma McMillan, Elizabeth Stevenson, Ewan Tempero, and Paul Denny. 2018. Ladebug: An online tool to help novice programmers improve their debugging skills. In *Proceedings of the 23rd Annual ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE '18)*. ACM, New York, NY, 159–164. DOI: https://doi.org/10.1145/3197091.3197098

[96] Andrew Luxton-Reilly, Simon, Ibrahim Albluwi, Brett A. Becker, Michail Giannakos, Amruth N. Kumar, Linda Ott, James Paterson, Michael James Scott, Judy Sheard, and Claudia Szabo. 2018. Introductory programming: A systematic literature review. In *Proceedings Companion of the 23rd Annual ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE '18 Companion)*. ACM, New York, NY, 55–106. DOI: https://doi.org/10.1145/3293881.3295779

[97] Nicholas Lytle, Mark Floryan, and Tiffany Barnes. 2019. Effects of a pathfinding program visualization on algorithm development. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education (SIGCSE '19)*. ACM, New York, NY, 225–231. DOI: https://doi.org/10.1145/3287324.3287391

[98] S. MacNeil, A. Tran, A. Hellas, J. Kim, S. Sarsa, Paul Denny, Seth Bernstein, and Juho Leinonen. 2023. Experiences from using code explanations generated by large language models in a web software development e-book. In *Proceedings of the 54th ACM Technical Symposium on Computer Science Education*, Vol. 1, 931–937.

[99] Katerina Mangaroska, Kshitij Sharma, Dragan Gašević, and Michail Giannakos. 2022. Exploring students' cognitive and affective states during problem solving through multimodal data: Lessons learned from a programming activity. *J. Comput. Assist. Learn.* 38, 1 (Feb. 2022), 40–59. DOI: https://doi.org/10.1111/jcal.12590

[100] Renée 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. *Comput. Sci. Educ.* 18, 2 (Jun. 2008), 67–92. DOI: https://doi.org/10.1080/08993400802114581

[101] Mary L. McHugh. 2012. Interrater reliability: The Kappa statistic. *Biochem. Med.* 22, 3 (2012), 276–282. DOI: https://doi.org/10.1016/j.jocd.2012.03.005

[102] Debra K. Meyer and Julianne C. Turner. 2006. Re-conceptualizing emotion and motivation to learn in classroom contexts. *Educ. Psychol. Rev.* 18, 4 (Dec. 2006), 377–390. DOI: https://doi.org/10.1007/s10648-006-9032-1

[103] Tilman Michaeli and Ralf Romeike. 2019. Current status and perspectives of debugging in the K12 classroom: A qualitative study. In *Proceedings of the 2019 IEEE Global Engineering Education Conference (EDUCON '19)*. IEEE. DOI: https://doi.org/10.1109/educon.2019.8725282

[104] 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 (WiPSCE'19, Article 15)*. ACM, New York, NY, 1–7. DOI: https://doi.org/10.1145/3361721.3361724

[105] Michael A. Miljanovic and Jeremy S. Bradbury. 2017. RoboBUG: A serious game for learning debugging techniques. In *Proceedings of the 2017 ACM Conference on International Computing Education Research (ICER '17)*. ACM, New York, NY, 93–100. DOI: https://doi.org/10.1145/3105726.3106173

[106] Andrés Moreno, Niko Myller, Erkki Sutinen, and Mordechai Ben-Ari. 2004. Visualizing programs with Jeliot 3. In *Proceedings of the Working Conference on Advanced Visual Interfaces (AVI '04)*. ACM, New York, NY, 373–376. DOI: https://doi.org/10.1145/989863.989928

[107] Laurie Murphy, Sue Fitzgerald, Brian Hanks, and Renée McCauley. 2010. Pair debugging: A transactive discourse analysis. In *Proceedings of the 6th International Workshop on Computing Education Research (ICER '10)*. ACM, New York, NY, 51–58. DOI: https://doi.org/10.1145/1839594.1839604

[108] 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. *SIGCSE Bull.* 40, 1 (Mar. 2008), 163–167. DOI: https://doi.org/10.1145/1352322.1352191

[109] Laurie Murphy and Lynda Thomas. 2008. Dangers of a fixed mindset: Implications of self-theories research for computer science education. In *Proceedings of the 13th Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE '08)*. ACM, New York, NY, 271–275. DOI: https://doi.org/10.1145/1384271.1384344

[110] 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 (ICER '17)*. ACM, New York, NY, 2–11. DOI: https://doi.org/10.1145/3105726.3106178

[111] Devon H. O'Dell. 2017. The debugging mindset: Understanding the psychology of learning strategies leads to effective problem-solving skills. *Queueing Syst.* 15, 1 (Feb. 2017), 71–90. DOI: https://doi.org/10.1145/3055301.3068754

[112] Paul W. Oman, Curtis, R. Cook, and Murthi Nanja. 1989. Effects of programming experience in debugging semantic errors. *J. Syst. Softw.* 9, 3 (Mar. 1989), 197–207. DOI: https://doi.org/10.1016/0164-1212(89)90040-X

[113] Matthew J. Page, Joanne E. McKenzie, Patrick M. Bossuyt, Isabelle Boutron, Tammy C. Hoffmann, Cynthia D. Mulrow, Larissa Shamseer, Jennifer M. Tetzlaff, Elie A. Akl, Sue E. Brennan, Roger Chou, Julie Glanville, Jeremy M. Grimshaw, Asbjørn Hróbjartsson, Manoj M. Lalu, Tianjing Li, Elizabeth W. Loder, Evan Mayo-Wilson, Steve McDonald, Luke A. McGuinness, Lesley A. Stewart, James Thomas, Andrea C. Tricco, Vivian A. Welch, Penny Whiting, and David Moher. 2021. The PRISMA 2020 statement: An updated guideline for reporting systematic reviews. *Rev. Esp. Cardiol.* 74, 9 (Sep. 2021), 790–799. DOI: https://doi.org/10.1016/j.rec.2021.07.010

[114] Ernesto Panadero, Anders Jonsson, and Juan Botella. 2017. Effects of self-assessment on self-regulated learning and self-efficacy: Four meta-analyses. *Educ. Res. Rev.* 22 (Nov. 2017), 74–98. DOI: https://doi.org/10.1016/j.edurev.2017.08.004

[115] Yulia Pechorina, Keith Anderson, and Paul Denny. 2023. Metacodenition: Scaffolding the problem-solving process for novice programmers. In *Proceedings of the 25th Australasian Computing Education Conference (ACE '23)*. ACM, New York, NY, 59–68. DOI: https://doi.org/10.1145/3576123.3576130

[116] D. N. Perkins and Fay Martin. 1986. Fragile knowledge and neglected strategies in novice programmers. In *Papers Presented at the 1st Workshop on Empirical Studies of Programmers on Empirical Studies Of Programmers*. Ablex Publishing Corp., 213–229.

[117] Michael Perscheid, Benjamin Siegmund, Marcel Taeumel, and Robert Hirschfeld. 2017. Studying the advancement in debugging practice of professional software developers. *Softw. Qual. Jo.* 25, 1 (Mar. 2017), 83–110. DOI: https://doi.org/10.1007/s11219-015-9294-2

[118] Chris Proctor. 2019. Measuring the computational in computational participation: Debugging interactive stories in middle school computer science. In *13th International Conference on Computer Supported Collaborative Learning (CSCL)* 1 (2019), 104–111.

[119] Adalbert Gerald Soosai Raj, Jignesh M. Patel, Richard Halverson, and Erica Rosenfeld Halverson. 2018. Role of live-coding in learning introductory programming. In *Proceedings of the 18th Koli Calling International Conference on Computing Education Research (Koli Calling '18, Article 13)*. ACM, New York, NY, 1–8. DOI: https://doi.org/10.1145/3279720.3279725

[120] Mitchel Resnick, John Maloney, Andrés Monroy-Hernández, Natalie Rusk, Evelyn Eastmond, Karen Brennan, Amon Millner, Eric Rosenbaum, Jay Silver, Brian Silverman, and Yasmin Kafai. 2009. Scratch. *Commun. ACM* 52, 11 (Nov. 2009), 60–67. DOI: https://doi.org/10.1145/1592761.1592779

[121] Dmitry Resnyansky, Mark Billinghurst, and Arindam Dey. 2019. An AR/TUI-supported debugging teaching environment. In *Proceedings of the 31st Australian Conference on Human-Computer-Interaction (OZCHI '19)*. ACM, New York, NY, 590–594. DOI: https://doi.org/10.1145/3369457.3369538

[122] Carol Rodgers. 2002. Defining reflection: Another look at John Dewey and reflective thinking. *Teach. Coll. Rec.* 104, 4 (Apr. 2002), 842–866. DOI: https://doi.org/10.1111/1467-9620.00181

[123] André L. Santos. 2018. Enhancing visualizations in pedagogical debuggers by leveraging on code analysis. In *Proceedings of the 18th Koli Calling International Conference on Computing Education Research (Koli Calling '18, Article 11)*. ACM, New York, NY, 1–9. DOI: https://doi.org/10.1145/3279720.3279732

[124] Dale H. Schunk and Peggy A. Ertmer. 2000. Self-regulation and academic learning: Self-efficacy enhancing interventions. In *Handbook of Self-Regulation*. Monique Boekaerts, Paul R. Pintrich, and Moshe Zeidner (Eds.), Academic Press, San Diego, 631–649. DOI: https://doi.org/10.1016/B978-012109890-2/50048-2

[125] P. A. Schutz and R. E. Pekrun. 2007. *Emotion in Education: A Volume in Educational Psychology*, Vol.348. Academic Press, Elsevier, Cambridge, MA, 3–10.

[126] Daniel L. Schwartz, Jessica M. Tsang, and Kristen P. Blair. 2016. *The ABCs of How We Learn: 26 Scientifically Proven Approaches, How They Work, and When to Use Them*. W. W. Norton & Company.

[127] Michael James Scott and Gheorghita Ghinea. 2014. Measuring enrichment: The assembly and validation of an instrument to assess student self-beliefs in CS1. In *Proceedings of the 10th Annual Conference on International Computing Education Research (ICER '14)*. ACM, New York, NY, 123–130. DOI: https://doi.org/10.1145/2632320.2632350

[128] Valerie J. Shute, Matthew Ventura, and Fengfeng Ke. 2015. The power of play: The effects of portal 2 and lumosity on cognitive and noncognitive skills. *Comput. Educ.* 80 (Jan. 2015), 58–67. DOI: https://doi.org/10.1016/j.compedu.2014.08.013

[129] Beth Simon, Dennis Bouvier, Tzu-Yi Chen, Gary Lewandowski, Robert McCartney, and Kate Sanders. 2008. Common sense computing (episode 4): Debugging. *Comput. Sci. Educ.* 18, 2 (Jun. 2008), 117–133. DOI: https://doi.org/10.1080/08993400802114698

[130] Arnan Sipitakiat and Nusarin Nusen. 2012. Robo-blocks: Designing debugging abilities in a tangible programming system for early primary school children. In *Proceedings of the 11th International Conference on Interaction Design and Children (IDC '12)*. ACM, New York, NY, 98–105. DOI: https://doi.org/10.1145/2307096.2307108

[131] Chrysanthos Socratous and Andri Ioannou. 2020. Common errors, successful debugging, and engagement during block-based programming using educational robotics in elementary education (June 2020). In *14th International Conference of the Learning Sciences (ICLS)* 2 (2020), 991–998.

[132] Elliot Soloway, Kate Ehrlich, and Jeffrey Bonar. 1982. Tapping into tacit programming knowledge. In *Proceedings of the 1982 Conference on Human Factors in Computing Systems (CHI '82)*. ACM, New York, NY, 52–57. DOI: https://doi.org/10.1145/800049.801754

[133] Juha Sorva, Ville Karavirta, and Lauri Malmi. 2013. A review of generic program visualization systems for introductory programming education. *ACM Trans. Comput. Educ.* 13, 4 (Nov. 2013), 1–64. DOI: https://doi.org/10.1145/2490822

[134] Gerry Stahl. 2005. Group cognition in computer-assisted collaborative learning. *J. Comput. Assist. Learn.* 21, 2 (Apr. 2005), 79–90. DOI: https://doi.org/10.1111/j.1365-2729.2005.00115.x

[135] Randy Stein and Susan E. Brennan. 2004. Another person's eye gaze as a cue in solving programming problems. In *Proceedings of the 6th International Conference on Multimodal Interfaces (ICMI '04)*. ACM, New York, NY, 9–15. DOI: https://doi.org/10.1145/1027933.1027936

[136] Ryo Suzuki, Gustavo Soares, Andrew Head, Elena Glassman, Ruan Reis, Melina Mongiovi, Loris D'Antoni, and Björn Hartmann. 2017. TraceDiff: Debugging unexpected code behavior using trace divergences. In *Proceedings of*

the 2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC '17). IEEE, 107–115. DOI: https://doi.org/10.1109/VLHCC.2017.8103457

[137] John Sweller. 1988. Cognitive load during problem solving: Effects on learning. *Cogn. Sci.* 12, 2 (Apr. 1988), 257–285. DOI: https://doi.org/10.1016/0364-0213(88)90023-7

[138] John Sweller, Jeroen J. G. van Merrienboer, and Fred G. W. C. Paas. 1998. Cognitive architecture and instructional design. *Educ. Psychol. Rev.* 10, 3 (Sep. 1998), 251–296. DOI: https://doi.org/10.1023/A:1022193728205

[139] Tamara van Gog and Nikol Rummel. 2010. Example-based learning: Integrating cognitive and social-cognitive research perspectives. *Educ. Psychol. Rev.* 22, 2 (Jun. 2010), 155–174. DOI: https://doi.org/10.1007/s10648-010-9134-7

[140] Iris Vessey. 1985. Expertise in debugging computer systems: A process analysis. *Int. J. Man. Mach. Stud.* 23, 5 (Nov. 1985), 459–494. DOI: https://doi.org/10.1016/S0020-7373(85)80054-7

[141] Ioannis Vourletsis, Panagiotis Politis, and Ilias Karasavvidis. 2021. The effect of a computational thinking instructional intervention on students' debugging proficiency level. In *Res. E-Learn. ICT Educ.* 15–34. DOI: https://doi.org/10.1007/978-3-030-64363-8_2

[142] L. S. Vygotsky and Michael Cole. 1978. *Mind in Society: Development of Higher Psychological Processes*. Harvard University Press.

[143] Jacqueline Whalley, Amber Settle, and Andrew Luxton-Reilly. 2021. Analysis of a process for introductory debugging. In *Proceedings of the Australasian Computing Education Conference (ACE '21)*. ACM, New York, NY, 11–20. DOI: https://doi.org/10.1145/3441636.3442300

[144] Jacqueline Whalley, Amber Settle, and Andrew Luxton-Reilly. 2021. Novice reflections on debugging. In *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education (SIGCSE '21)*. ACM, New York, NY, 73–79. DOI: https://doi.org/10.1145/3408877.3432374

[145] Claes Wohlin. 2014. Guidelines for snowballing in systematic literature studies and a replication in software engineering. In *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering (EASE '14, Article 38)*. ACM, New York, NY, 1–10. DOI: https://doi.org/10.1145/2601248.2601268

[146] Gary K. W. Wong and Shan Jiang. 2018. Computational thinking education for children: Algorithmic thinking and debugging. In *Proceedings of the 2018 IEEE International Conference on Teaching, Assessment, and Learning for Engineering (TALE '18)*. IEEE, 328–334. DOI: https://doi.org/10.1109/TALE.2018.8615232

[147] 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 (SIGCSE '18)*. ACM, New York, NY, 344–349. DOI: https://doi.org/10.1145/3159450.3159527

[148] Byung-Do Yoon and O. N. Garcia. 1998. Cognitive activities and support in debugging. In *Proceedings of the 4th Annual Symposium on Human Interaction with Complex Systems*. IEEE, 160–169. DOI: https://doi.org/10.1109/HUICS.1998.659974

[149] Baichang Zhong and Tingting Li. 2020. Can pair learning improve students' troubleshooting performance in robotics education? *ACM J. Educ. Resour. Comput.* 58, 1 (Mar. 2020), 220–248. DOI: https://doi.org/10.1177/0735633119829191

[150] Barry J. Zimmerman. 1990. Self-regulated learning and academic achievement: An overview. *Educ. Psychol.* 25, 1 (1990), 3–17.

Appendix

A Complete List of Included Papers

Table A1. Full List of Papers in Our Sample, with Paper Name, Authors, Year, Subject Education Level, Programming Language, and Paper Reference

Paper	Authors	Year	Subject Education	Programming Language	Ref
"Reflective Debugging in Spinoza V3.0."	Abu Deeb and Hickey	2021	University	Python	[1]
"Characterizing the Pedagogical Benefits of Adaptive Feedback for Compilation Errors by Novice Programmers."	Ahmed et al.	2020	University	C	[3]

(Continued)

Table A1. Continued

"Unplugged Debugging Activities for Developing Young Learners' Debugging Skills."	Ahn et al.	2022	Elementary	Block-based	[4]
"Evaluating the Use of Pedagogical Virtual Machine with Augmented Reality to Support Learning Embedded Computing Activity."	Alrashidi et al.	2017	University	Python	[8]
"Reusing Bugged Source Code to Support Novice Programmers in Debugging Tasks."	Ardimento et al.	2019	University	Java	[10]
"The Effect of Play and Worked Examples on First and Third Graders' Creating and Debugging of Programming Algorithms."	Boffer-ding et al.	2022	Elementary	Other	[22]
"Debugging Students' Debugging Process."	Böttcher et al.	2016	University	Java	[24]
"Its Debug: Practical Results"	Carter	2015	University, High-school	Java	[28]
"Learning From Worked Examples, Erroneous Examples, and Problem Solving: Toward Adaptive Selection of Learning Activities."	Chen et al.	2020	University	Other	[30]
"Computational Thinking in Augmented Reality: An Investigation of Collaborative Debugging Practices."	Chung et al.	2020	University	Block-based	[32]
"Dynamic Program Visualizations: an Experience Report."	Cross et al.	2014	University	Java	[33]
"Art as a Point of Departure for Understanding Student Experience in Learning to Code."	Dahn et al.	2020	Middle-school, High-school	Java, Block-based, Other	[34]
"From Video Games to Debugging Code."	Deitz et al.	2016	University	C	[35]
"Debugging as a Context for Fostering Reflection on Critical Thinking and Emotion."	DeLiema et al.	2019	Middle-school, High-school	Java, Block-based, Other	[36]
"Defining and Supporting a Debugging Mindset in Computer Engineering Courses."	Duwe et al.	2022	University	Hardware, C	[42]

(Continued)

Table A1. Continued

"Turn up the Heat! Using Heat Maps to Visualize Suspicious Code to Help Students Successfully Complete Programming Problems Faster."	Edmison and Edwards	2020	University	Java	[46]
"Program Slicing Technique: A Novel Approach to Improve Programming Skills in Novice Learners."	Eranki et al.	2016	University	Java	[47]
"Using Quicksand to Improve Debugging Practice in Post-Novice Level Students."	Fenwick et al.	2012	University	C	[51]
"Debugging by Design: A Constructionist Approach to High School Students' Crafting and Coding of Electronic Textiles as Failure Artefacts."	Fields et al.	2021	High-school	C++, Hardware	[52]
"Read the Debug Manual: A Debugging Manual for CS1 Students."	Garcia et al.	2022	University	C++	[54]
"Effects of Hints on Debugging Scratch Programs: An Empirical Study with Primary School Teachers in Training."	Greifenstein et al.	2021	Adult	Block-based	[58]
"Jype - a Program Visualization and Programming Exercise Tool for Python."	Helminen et al.	2010	University	Python	[67]
"The Effects of Adaptive Procedural Levels on Engagement and Performance in an Educational Programming Game."	Jemmali et al.	2022	Adult	Custom	[71]
"Teaching Explicit Programming Strategies to Adolescents."	Ko et al.	2019	Middle-school, High-school	Javascript	[80]
"Personifying Programming Tool Feedback Improves Novice Programmers' Learning."	Lee et al.	2011	Adult	Custom	[85]
"Understanding Problem Solving Behavior of 6–8 Graders in a Debugging Game."	Liu et al.	2017	Middle-school	Block-based	[91]
"Ladebug: An Online Tool to Help Novice Programmers Improve Their Debugging Skills."	Luxton-Reilly et al.	2018	University	Python	[95]
"Effects of a Pathfinding Program Visualization on Algorithm Development."	Lytle et al.	2019	University	Java	[97]

(Continued)

Table A1. Continued

"Improving Debugging Skills in the Classroom: The Effects of Teaching a Systematic Debugging Process."	Michaeli and Romeike	2019	Middle-school, High-school	Java, Other	[104]
"RoboBUG: A Serious Game for Learning Debugging Techniques."	Miljanovic et al.	2017	University	C++	[105]
"Pair Debugging: A Transactive Discourse Analysis."	Murphy et al.	2010	University	Java	[107]
"Measuring the Computational in Computational Participation: Debugging Interactive Stories in Middle School Computer Science."	Proctor	2019	Middle-school	Other	[118]
"Role of Live-Coding in Learning Introductory Programming."	Raj et al.	2018	University	C	[119]
"An AR/TUI-Supported Debugging Teaching Environment."	Resnyansky et al.	2019	none	Block-based, Tangibles	[121]
"Enhancing Visualizations in Pedagogical Debuggers by Leveraging on Code Analysis."	Santos	2018	University	Java	[123]
"Robo-Blocks: Designing Debugging Abilities in a Tangible Programming System for Early Primary School Children."	Sipitakiat et al.	2012	Elementary	Tangibles	[130]
"Common Errors, Successful Debugging, and Engagement During Block-Based Programming Using Educational Robotics in Elementary Education."	Socratous et al.	2020	Elementary	Block-based	[131]
"TraceDiff: Debugging Unexpected Code Behavior Using Trace Divergences."	Suzuki et al.	2017	University	Python	[136]
"The Effect of a Computational Thinking Instructional Intervention on Students' Debugging Proficiency Level and Strategy Use."	Vourletsis et al.	2021	Middle-school	Block-based	[141]
"Analysis of a Process for Introductory Debugging."	Whalley et al.	2021	University	Python	[143]
"Novice Reflections on Debugging."	Whalley et al.	2021	University	Python	[144]

(Continued)

Table A1. Continued

"Computational Thinking Education for Children: Algorithmic Thinking and Debugging."	Wong et al.	2018	Elementary	Block-based	[146]
"Can Pair Learning Improve Students' Troubleshooting Performance in Robotics Education?"	Zhong et al.	2020	High-school	C++	[149]

Received 4 January 2024; revised 9 July 2024; accepted 19 July 2024