Proceedings of the Thirty-Fourth International Joint Conference on Artificial Intelligence (IJCAI-25)
Sister Conferences Best Papers Track

How to Teach Programming in the AI Era? Using LLMs as a Teachable Agent for Debugging (Extended Abstract)[1]

Qianou Ma[1] , **Hua Shen**[2] , **Ken Koedinger**[1] and **Tongshuang Wu**[1]
[1]Carnegie Mellon University
[2]University of Washington
{qianoum, krk, sherryw}@cs.cmu.edu, huashen@uw.edu

Abstract

Large Language Models (LLMs) excel at *generating* content at impeccable speeds. However, they are imperfect and still make various mistakes. In Computer Science education, as LLMs are widely recognized as "AI pair programmers," it becomes increasingly important to train students on *evaluating* and *debugging* LLM-generated codes. In this work, we introduce HYPOCOMPASS, a novel system to facilitate deliberate practice on debugging, where human novices play the role of Teaching Assistants and help LLM-powered teachable agents debug code. We enable effective task delegation between students and LLMs in this learning-by-teaching environment: students focus on *hypothesizing the cause of code errors*, while adjacent skills like code completion are offloaded to LLM-agents. Our evaluations demonstrate that HYPOCOMPASS generates high-quality training materials (e.g., bugs and fixes), outperforming human counterparts fourfold in efficiency, and significantly improves student performance on debugging by 12% in the pre-to-post test.

1 Introduction

LLMs are becoming integral to software development — commercialized tools like GitHub Copilot are advertised as "your AI pair programmer" and generate up to 46% of users' code [Dohmke, 2023]. Despite their prevalence, LLMs often produce unpredictable mistakes [Ganguli *et al.*, 2022], e.g., GPT-4 can still make mistakes 17% of the time in coding tasks for introductory and intermediate programming courses [Savelka *et al.*, 2023]. The impressive yet imperfect generative capabilities of LLMs, coupled with the associated risks of excessive reliance on these models, underscore the importance of teaching students *evaluation*, or debugging and testing skills [Becker *et al.*, 2023] for programming.

However, debugging tends to be overlooked in curricula, especially in introductory Computer Science classes (i.e., CS1) [News, 2014], as instructors have limited time budget for developing specialized debugging materials and assessments [McCauley *et al.*, 2008]. Consequently, students invest substantial time and effort in *hypothesizing* the cause of bugs while grappling with other cognitively demanding tasks, such as understanding and writing code. These challenges prompt us to ask:

Research Question: Can we train students to improve debugging skills by providing *explicit* and *scaffolded* practice *with minimal cost to instructor time?*

In this work[1], we focus on training students' abilities in *hypothesis construction*, a critical step in debugging as established by prior work [Xu and Rajlich, 2004; Zeller, 2009]. We introduce HYPOCOMPASS (Figure 1), a LLM-augmented intelligent tutoring system for debugging. We have LLMs imitate CS1 students who have written buggy codes, and human novice students assume the role of Teaching Assistants (TAs). This enables students to deliberately practice the skill of *hypothesizing* about the defects of LLM-generated code, delegating other tasks not core to hypothesis construction (e.g., code completion) to the LLM. As a result, HYPOCOMPASS fosters engaging learning using the *teachable agent* framework [Blair *et al.*, 2007] and provides students with guided exposure to LLM-generated bugs. We also employ strategies such as task formation and over-generate-then-select to improve LLM generation quality (Section 2.1).

We conducted two evaluation studies and found that HYPOCOMPASS *saves instructors' time in material generation* and is *beneficial to student learning*. In our LLM evaluation (Section 3), HYPOCOMPASS achieved a 90% success rate in generating and validating a complete set of materials, *four times faster than human generation.* Our learning evaluation with 19 novices (Section 4) showed that HYPOCOMPASS significantly improved students' pre-to-post test performance by 12% and decreased their completion time by 14%.

In summary, we contribute:

- A pragmatic solution that balances the benefits and risks of LLMs in learning. We prepare students to engage with imperfect LLMs, and we highlight the importance of *role-playing* for practical LLM application and *task delegation* to help students focus on essential skills.
- A theoretically grounded instructional design for debugging. To the best of our knowledge, we are the first to provide aligned instruction and assessment on hypothesis construction – forming hypotheses about the source of error, a core bottleneck in debugging [Whalley *et al.*, 2021].

[1]Original Paper: [Ma *et al.*, 2024]

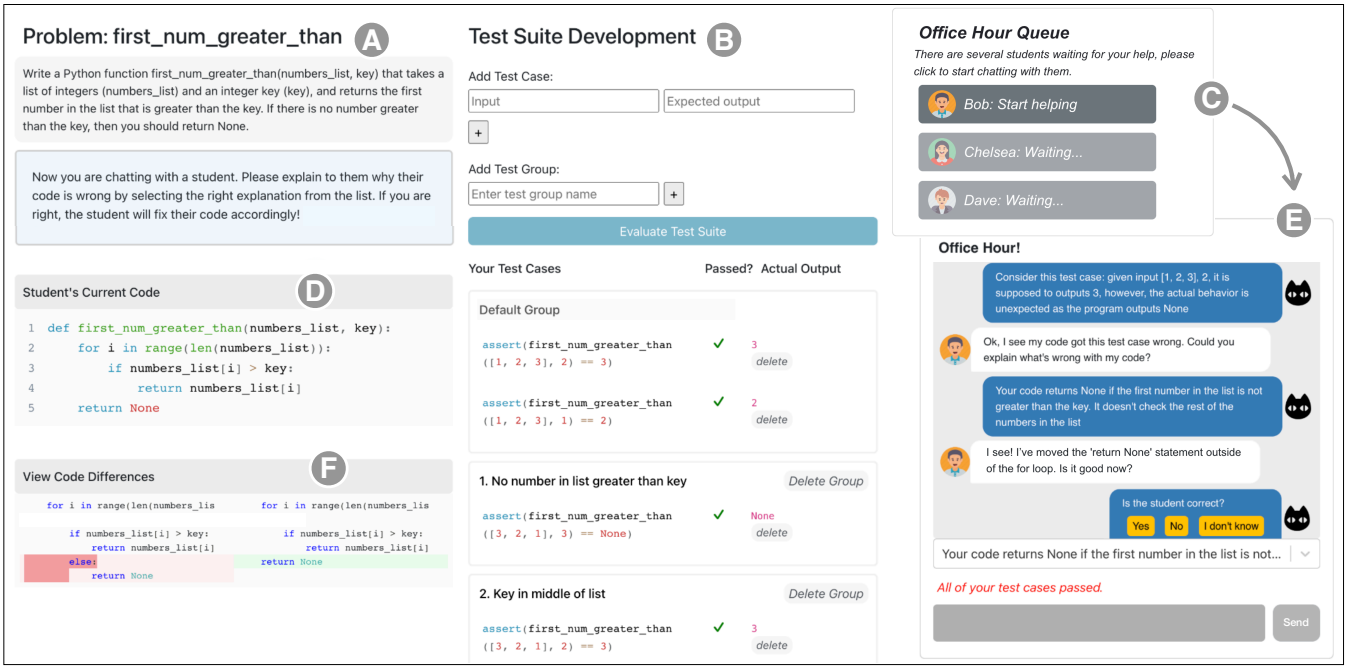

Figure 1: In HYPOCOMPASS, given a problem description (A), a student user (in the role of a Teaching Assistant) needs to compile a test suite (B) and assist multiple LLM-simulated agents (e.g., *Bob, Chelsea, Dave*) in an Office Hour Queue (C) through a chat interface (E). Each LLM-agent acts as a novice seeking help with a buggy solution (D) and provides feedback to the user (F).

2 The Design of HYPOCOMPASS

Grounded in the cognitive process [Xu and Rajlich, 2004] and the novice-expert difference in hypothesis-driven debugging [Edwards and Shams, 2014], we specify two crucial learning components for HYPOCOMPASS: comprehensive and accurate hypothesis construction. Prior work shows that hypothesis construction is closely connected with testing [Zeller, 2009]: each additional test case should, ideally, be a hypothesis about what can go wrong in the program. In turn, a *comprehensive* test suite (i.e., a set of test cases) should allow an effective debugger to construct a *accurate* hypothesis about why the program is wrong. We thus design toward two learning objectives (Figure 2A,D):

LO1 **Comprehensive Hypothesis Construction**: Construct a comprehensive test suite that well covers the possible errors for the given problem.

LO2 **Accurate Hypothesis Construction**: Given the failed test cases, construct an accurate explanation of how the program is wrong.

Interface and Key Components. In the HYPOCOMPASS interface (Figure 1), a human student would be asked to play the role of a TA where they help an LLM-simulated student (LLM-agent) in debugging. They begin by constructing test suites that represent different hypotheses about possible bugs, and then use these to help each agent debug code via a dialog interface. Each agent presents a buggy code snippet; students provide tests and select from candidate explanations. If correct, the LLM-agent revises the code and provides feedback; if incorrect, the agent highlights the mismatch to prompt reflection. Once all bugs are fixed, students move on to the next agent. A typical session includes two programming problems, each with three buggy agents.

We emphasize two core interaction components:

- **Role-play with imperfect LLMs.** LLMs act as novice students submitting buggy code, while human learners assist them. This teachable agent setup encourages reflection, builds confidence, and more importantly provides *guided exposure to realistic LLM errors* [Shahriar and Matsuda, 2023; Blair *et al.*, 2007].
- **Focused task delegation.** Students concentrate on hypothesis construction –completing test suites (LO1) and mapping explanations to bugs (LO2) – while LLMs handle code generation, scaffolding, and feedback. This separation supports deliberate, guided practice on the learning objectives, and align student interaction flow with the cognitive model [Xu and Rajlich, 2004] (Figure 2B, C_1).

2.1 LLM Integration

We use LLM to generate five types of materials: (1) test case category hints, (2) test case hints, (3) buggy programs, (4) explanations of bugs, and (5) programs with bugs fixed. We reduce instructor workload by generating practices using just a problem description, a reference solution, and a test suite with about 10 inputs, and we further minimize human verification overhead with optimized prompts and automated algorithms (example prompts in Section 2.1)[2]. Below are key factors to the success of generation:

[2]Full prompts and parameters are in Supplements.

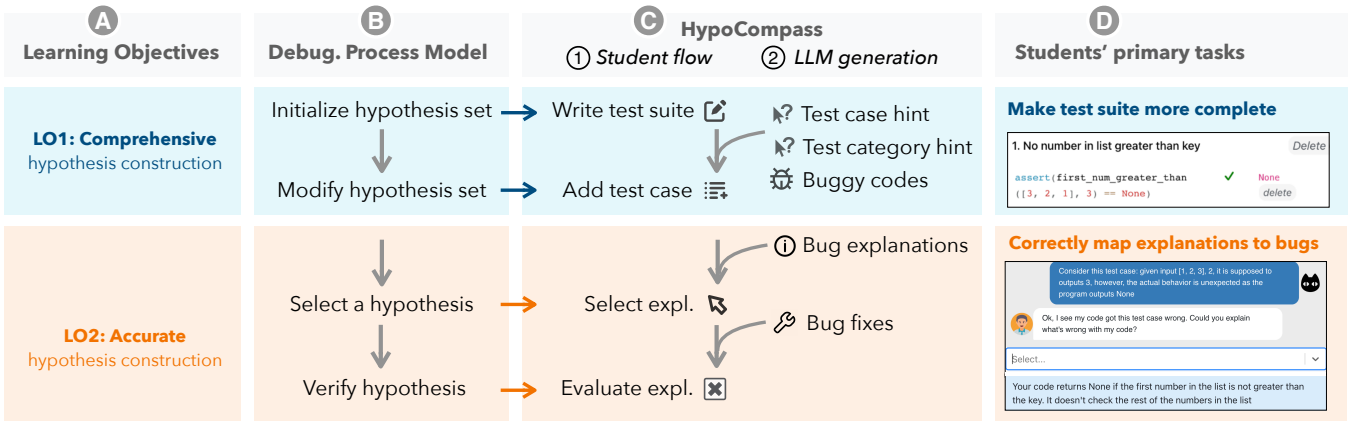

Figure 2: To enable deliberate practice, we establish a close mapping between the (A) learning objectives, (B) the cognitive debugging process model, (C) the HypoCompass interaction flow, and (D) the primary tasks students perform in HypoCompass. We offload various material generation tasks to LLMs (C_2).

Task Formation and Decomposition. To align LLM behavior with intended tasks [Xie *et al.*, 2023], we reformulate prompts to improve reliability. For instance, instead of asking for *local bug fixes* — which often leads to over-correcting multiple bugs in one go — we reframe it as a *code translation* task (`old → new snippet`), reducing over-fix errors by 70%. For complex tasks like local bug fixing, we use LLM-chains [Wu *et al.*, 2022] to break tasks into smaller, stable steps. We also use hierarchical prompt design, prioritizing core requirements (e.g., bug identification) over secondary ones (e.g., word limits), improving success rates by 40%.

Over-Generate-then-Select. LLMs can generate diverse outputs, but ensuring pedagogical value requires careful selection. For example, it is hard to enforce LLMs to generate behaviorally distinct bugs through prompting, as it requires LLMs to "know" bug behaviors. Nonetheless, we can over-generate multiple solutions with mixed qualities [MacNeil *et al.*, 2022], and then select a subset of desired ones as follows:

1. We over-generate codes, discard correct ones, and select a diverse subset based on maximum Euclidean distance on error vectors.
2. We select distractor explanations and feedback by finding buggy codes with behaviors closest to the target (smallest Euclidean distance) and using their explanations.
3. We cluster over-generated test cases using hierarchical agglomerative clustering [Lukasová, 1979] to select meaningful test category hints.

Human-in-the-Loop Verification. To avoid error propagation in sequential generation, we include human verification at each step from buggy code to explanations. Editing times and verification details are in [Ma *et al.*, 2024].

3 LLM Evaluation: Efficiency and Quality

We evaluated the generations on six different problems from prior work [Dakhel *et al.*, 2023] and our own problems (detailed in Table 4 in Supplements). On average, for each problem, we generated 3 test category hints, 10 test case hints, 24 buggy programs, explanation and fix instructions, and 33 bug fixes. The total number, success rates, and success criteria are summarized in [Ma *et al.*, 2024].

Method. Two authors annotated 10% of the generations at each step individually, and discussed to resolve the disagreement and update the codebook. An external instructor annotated the same 10% of LLM-generated materials, using the updated codebook. We calculated the inter-rater reliability (IRR) between the external instructor and the resolved annotation among the two authors using percent IRR and Cohen's Kappa. The agreements are satisfactory across different model generations (IRR% $> 90\%$ and $\kappa > 0.75$). One author annotated the rest of the materials to calculate the success rates. We log the verification and editing *time*, as proxies to the instructor overhead.

To compare LLM and human generations, we recruited two experienced CS TAs to each create practice materials. Each TA received the same input as LLMs, was asked to produce one set of materials matching the amount of content LLMs produced, and was compensated for their time.

Result: Efficient and High-Quality Generation. We achieve high-quality generation: a complete set of practice materials with 9 buggy programs (3 for practice and 6 more as distractors), 9 bug explanations, 9 bug fixes, 10 test case hints, and 3 test category hints can be generated with a 90% success rate and only takes 15 minutes to label and edit. As we *over-generate* and automatically select buggy code, a success rate over 50% is reasonable for practical use.

Employing LLMs can also be significantly more efficient. In total, a TA spent around 60 minutes to generate one set of practice materials for HypoCompass. One TA noted the difficulty in consistently creating unique and high-quality materials after 30 minutes: *"the importance of the bug I create would start to decline."* The same author evaluated the TAs' generations using the annotation codebook, which had a 100% success rate and took 11 minutes. The time spent to generate and edit instructional materials for HypoCompass using LLMs was 4.67 times less than that of the human TAs.

Material	Generation goal	Temp.
Bug expl. & fix instruct.	To describe each unique bug and write a fix. If there are multiple bugs in the code, generate explanations and fixes separately.	0.3
[User]	Hi, I'm a student in your class. I'm having trouble with this problem in the programming assignment: {problem_description} Here's my buggy code: {buggy_code} What's wrong with my code? **List all the unique bugs included, but do not make up bugs.** For each point, put in the format of: {explanation: accurate and concise explanation of what the code does and what the bug is, for a novice, fix: how to fix the bug, within 30 words} Only **return the bullet list.** Do not write any other text or code.	
Bug fix	To edit the buggy code according to the fix instruction, w/o over- or under- fix.	0.3
[User]	Original code: {buggy_code}; Code modification: {explanation}; **Translate the statement into actual, minimal code change in this format:** {original code snippet: ""copy the lines of code that need editing"" -> edited code snippet: ""write the edited code snippet""} [LLM] {old to new snippet in JSON, e.g., numbers_list[i] <= key → numbers_list[i] > key } [User] Old Code:{buggy_code}; Instruction:{Old snippet to new snippet}; New Code:	

Table 1: Prompts and temperatures (Temp.) for generating bug explanations, and fixes.

4 Learning Evaluation: Pre- / Post-Test Study

Can novices better formulate hypotheses after engaging with HYPOCOMPASS*?* We conducted a learning evaluation with 19 students and compared the difference in speed and performance from the pre-test to the post-test.

Method: Study Procedure and Participants. We conducted a one-hour study with 19 students from four U.S. institutions (12 females; 8 non-native English speakers; average age 20.7). After a screening exercise to ensure a suitable skill range, participants completed a pre-survey, a 20-minute pre-test, interacted with HYPOCOMPASS on two debugging problems, then took a 20-minute post-test and a final survey. One problem was reused from the pre-test to control for comprehension. Participants received a $15 gift card.

Quantitative Result: Learning Gains. A two-tailed paired t-test showed that students' pre-test to post-test scores significantly improved by 11.7% ($p = 0.033 < 0.05$), and the time of completion significantly reduced by 13.6% ($p = 0.003$), indicating success in learning through HYPOCOMPASS. Note that the bugs used in pre-post tests are generated by humans and are not the same as in HYPOCOMPASS. As such, the significant learning gains indicate that students could learn debugging skills *transferable* to real-world bugs.

Where does the learning gain come from? We break down the analyses by learning objectives. We found a small 6.1% improvement in the score and a large 23.6% time reduction for *comprehensive hypothesis construction* (LO1), and a large 15.8% improvement in the score and a small 9.0% time reduction for *accurate hypothesis construction* (LO2). Therefore, students showed more efficiency enhancement in LO1, and more learning gains in LO2. Note that these improvements may confound with problem difficulty, as the items corresponding to LO1 (pre-test $\mu = 54\%$) seem easier than the ones for LO2 (pre-test $\mu = 38\%$).

Qualitative Result: Student Perceptions. We further unpack how HYPOCOMPASS contributed to learning by analyzing the survey responses. Students valued being able to offload some debugging subtasks to HYPOCOMPASS, such as writing code and explanations. For example, S1 said *"looking at the test behavior and the explanation options really helps relieve that burden."* Students also generally felt that the LLM-generated bugs and fixes were authentic. Most participants could not tell if their practiced programs were written by students or AI because of their experiences making or seeing similar mistakes from peers.

Moreover, students reported that HYPOCOMPASS was engaging and helped build confidence in debugging. A Wilcoxon signed-rank test shows a significant increase in self-rated confidence in debugging by 15% ($p = 0.007$). Students rated HYPOCOMPASS as significantly more engaging (6.0 out of 7), fun (6.0), and less frustrating (2.5) than their conventional way of learning debugging and testing ($p < 0.005$ for each). S8 especially liked the teachable agent setup: *"the role play just feels more natural because it feels like explaining to a rubber duck instead of talking to myself"*.

5 Discussion

Teachable Agent for Appropriate Reliance with Imperfect AIs. Our work illustrates a scenario in which *LLM-generated bugs are not seen as problems but rather as features.* HYPOCOMPASS's teachable agent setup provides students with *moderated exposure to imperfect LLMs*, and may help them learn that LLMs are fallible and calibrate trust accordingly. Future iterations could remove material validation and allow direct exposure to unfiltered LLM mistakes in real-time interactions, taking full advantage of the teachable agent framework. Students will naturally expect that the LLM-agent seeking help may make mistakes (e.g., fail to follow bug-fixing explanations). This approach, however, requires a more sophisticated design for scaffolding students in recognizing LLM errors.

Task Delegation for Shifting Learning Focus. Our exploration lays the foundation for a paradigm shift toward cultivating higher-order evaluation skills in the generative AI era. Essentially, we asked: what skills should we offload, and what should we learn? Most students in our study appreciated offloading subtasks to LLM (Section 4); however, some need more scaffolds, while others prefer less. Future research can investigate more personalized task delegation. For example, students who need more help can use LLMs to facilitate code tracing, and students can also write their own explanations for bugs based on their proficiency. Deciding the bare minimum programming skills and human-AI collaboration skills to teach also warrants further exploration [Ma *et al.*, 2023].

Acknowledgments

Thanks to the participants, reviewers, Vicky Zhou, Kelly Rivers, Michael Taylor, Michael Hilton, Michael Xieyang Liu, Kexin Yang, Jionghao Lin, Erik Harpstead, and other Ken's lab members for insights and help. Thanks to the gift funds from Adobe, Oracle, and Google; Thanks to the National Science Foundation (award CNS-2213791) for partial support of this work.

References

[Becker *et al.*, 2023] Brett A Becker, Paul Denny, James Finnie-Ansley, Andrew Luxton-Reilly, James Prather, and Eddie Antonio Santos. Programming is hard-or at least it used to be: Educational opportunities and challenges of ai code generation. In *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1*, pages 500–506, 2023.

[Blair *et al.*, 2007] Kristen Blair, Daniel L Schwartz, Gautam Biswas, and Krittaya Leelawong. Pedagogical agents for learning by teaching: Teachable agents. *Educ. Technol. Res. Dev.*, 47(1):56–61, 2007.

[Dakhel *et al.*, 2023] Arghavan Moradi Dakhel, Vahid Majdinasab, Amin Nikanjam, Foutse Khomh, Michel C Desmarais, and Zhen Ming Jack Jiang. Github copilot ai pair programmer: Asset or liability? *Journal of Systems and Software*, 203:111734, 2023.

[Dohmke, 2023] Thomas Dohmke. GitHub copilot x: The AI-powered developer experience. https://github.blog/2023-03-22-github-copilot-x-the-ai-powered-developer-experience/, March 2023. Accessed: 2023-9-5.

[Edwards and Shams, 2014] Stephen H Edwards and Zalia Shams. Do student programmers all tend to write the same software tests? In *Proceedings of the 2014 conference on Innovation & technology in computer science education*, ITiCSE '14, pages 171–176, New York, NY, USA, June 2014. Association for Computing Machinery.

[Ganguli *et al.*, 2022] Deep Ganguli, Danny Hernandez, Liane Lovitt, Amanda Askell, Yuntao Bai, Anna Chen, Tom Conerly, Nova Dassarma, Dawn Drain, Nelson Elhage, et al. Predictability and surprise in large generative models. In *Proceedings of the 2022 ACM Conference on Fairness, Accountability, and Transparency*, pages 1747–1764, 2022.

[Lukasová, 1979] Alena Lukasová. Hierarchical agglomerative clustering procedure. *Pattern Recognition*, 11(5-6):365–381, 1979.

[Ma *et al.*, 2023] Qianou Ma, Tongshuang Wu, and Kenneth Koedinger. Is AI the better programming partner? Human-Human pair programming vs. Human-AI pAIr programming. *arXiv preprint arXiv:2306.05153*, 2023.

[Ma *et al.*, 2024] Qianou Ma, Hua Shen, Kenneth Koedinger, and Tongshuang Wu. How to teach programming in the AI era? using llms as a teachable agent for debugging. *International Conference on Artificial Intelligence in Education (AIED)*, 2024.

[MacNeil *et al.*, 2022] Stephen MacNeil, Andrew Tran, Dan Mogil, Seth Bernstein, Erin Ross, and Ziheng Huang. Generating diverse code explanations using the gpt-3 large language model. In *Proceedings of the 2022 ACM Conference on International Computing Education Research-Volume 2*, pages 37–39, 2022.

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

[News, 2014] Ycombinator Hacker News. Why don't schools teach debugging? https://news.ycombinator.com/item?id=7215870, February 2014. Accessed: 2023-9-8.

[Savelka *et al.*, 2023] Jaromir Savelka, Arav Agarwal, Marshall An, Chris Bogart, and Majd Sakr. Thrilled by your progress! large language models (gpt-4) no longer struggle to pass assessments in higher education programming courses. *arXiv preprint arXiv:2306.10073*, 2023.

[Shahriar and Matsuda, 2023] Tasmia Shahriar and Noboru Matsuda. What and how you explain matters: Inquisitive teachable agent scaffolds knowledge-building for tutor learning. In *International Conference on Artificial Intelligence in Education*, pages 126–138. Springer, 2023.

[Whalley *et al.*, 2021] Jacqueline Whalley, Amber Settle, and Andrew Luxton-Reilly. Analysis of a process for introductory debugging. In *Proceedings of the 23rd Australasian Computing Education Conference*, ACE '21, pages 11–20. Association for Computing Machinery, March 2021.

[Wu *et al.*, 2022] Tongshuang Wu, Michael Terry, and Carrie Jun Cai. AI chains: Transparent and controllable Human-AI interaction by chaining large language model prompts. In *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems*, number Article 385 in CHI '22, pages 1–22, New York, NY, USA, 2022. Association for Computing Machinery.

[Xie *et al.*, 2023] Jian Xie, Kai Zhang, Jiangjie Chen, Renze Lou, and Yu Su. Adaptive chameleon or stubborn sloth: Unraveling the behavior of large language models in knowledge clashes, 2023.

[Xu and Rajlich, 2004] Shaochun Xu and V Rajlich. Cognitive process during program debugging. In *Proceedings of the Third IEEE International Conference on Cognitive Informatics, 2004.*, pages 176–182. IEEE, August 2004.

[Zeller, 2009] Andreas Zeller. *Why programs fail: A guide to systematic debugging*. Morgan Kaufmann, Oxford, England, 2 edition, June 2009.