A COMPUTATIONAL FRAMEWORK FOR SOCRATIC DEBUGGING CONVERSATIONS

by

Erfan Al-Hossami

A dissertation submitted to the faculty of
The University of North Carolina at Charlotte
in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in
Computing and Information Systems

Charlotte

2025

Approved by:

Dr. Razvan Bunescu

Dr. Mohsen Dorodchi

Dr. Samira Shaikh

Dr. Sandeep Kuttal

Dr. Tomek Strzalkowski

Dr. Meera Sridhar

ABSTRACT

ERFAN AL-HOSSAMI. A computational framework for Socratic debugging conversations.
(Under the direction of DR. RAZVAN BUNESCU)

The Socratic teaching method encourages students to solve problems through instructor-guided questioning, rather than providing direct answers. Although this method can enhance learning outcomes, it is both time-consuming and cognitively demanding, limiting instructors' ability to provide individualized attention at scale. Automated Socratic conversational agents offer a promising avenue for supplementing human instruction in programming education, yet their development has been constrained by the lack of appropriate datasets, evaluation frameworks, and principled approaches to dialogue generation. This dissertation presents a computational framework for automated Socratic debugging conversations in novice programming environments. The framework makes three important, interconnected contributions: (1) benchmarks and evaluation standards for Socratic debugging, (2) automated mining of student misconceptions from code submissions, and (3) generation of Socratic dialogue that guides students to discover and correct their errors. First, I introduce the novel task of Socratic debugging and present a benchmark dataset of expert-crafted multi-turn Socratic conversations, which has been used to evaluate various large language models in zero-shot and fine-tuned settings. Second, I describe an automated approach for mining known as well as novel student misconceptions in code submissions, which can provide crucial knowledge for targeted pedagogical interventions. Third, I introduce the concept of Reasoning Trajectories as intermediate representations of Socratic conversations that are designed to guide the student towards statements about code behavior that contradict their misconceptions. The ensuing cognitive dissonance is expected to lead to enduring belief updates that fix the misconception. Overall, the three contributions establish conceptual and computational foundations for automated Socratic agents. While the focus is on programming education, the framework described in this dissertation is generalizable to any domain that can benefit from Socratic teaching of problem-solving skills through guided discovery and correction of misconceptions. Furthermore, this work opens avenues for research on the optimization of personalized Socratic agents.

DEDICATION

To my grandfather, Erfan Azmeh, whose unfinished journey I now complete.

ACKNOWLEDGEMENTS

I am deeply grateful to my advisor, Dr. Razvan Bunescu, for his invaluable guidance, patience, and unwavering support throughout my doctoral journey. His mentorship has profoundly shaped my growth as a researcher.

I am especially grateful to Dr. Mohsen Dorodchi and Dr. Samira Shaikh, who mentored me during my undergraduate years and the early stages of my PhD program. Their guidance laid the foundation for my research journey and helped shape my interests in computing education and natural language processing. I would also like to extend my sincere appreciation to the other members of my dissertation committee, Dr. Sandeep Kuttal, Dr. Tomek Strzalkowski, and Dr. Meera Sridhar, for their thoughtful feedback and insights that strengthened this work.

I would like to thank my co-authors and all the data contributors to the Socratic Debugging dataset. Special thanks to Justin Smith, Ryan Teehan, Laurel Powell, Khyati Mahajan, Sandra Wiktor, Anusha Reddy, and Dr. Qiong Cheng for their considerable time and effort in contributing dialogues to the benchmark dataset. I also acknowledge Ilan Aktanova and Frank Garcia for their contributions to the programming exercises used in the dataset.

This research was supported in part by the United States Air Force (USAF) under Contract No. FA8750-21-C-0075 and the Graduate Assistant Support Plan (GASP) at UNC Charlotte. The project has also benefited from the Microsoft Accelerating Foundation Models Research (AFMR) grant program. I am grateful to Dr. Srijan Das for providing access to GPT models through the NAIRR Pilot initiative provided through NSF under grant No. NAIRR240338. Any opinions, findings, conclusions, or recommendations expressed in this material are

those of the author and do not necessarily reflect the views of the USAF, AFMR, or NSF.

I am deeply grateful to Abraham Sanders, my grandmother Salwa Al-Jadda, and the rest of my family, whose unwavering encouragement and support were critical to the completion of this dissertation.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

AI	An acronym for Artificial Intelligence.
AST	An acronym for Abstract Syntax Tree.
GI	An acronym for Guided Inquiry.
GPT	An acronym for Generative Pre-trained Transformer.
ITS	An acronym for Intelligent Tutoring System.
LLM	An acronym for Large Language Model.
LM	An acronym for Language Model.
NLG	An acronym for Natural Language Generation.
NLP	An acronym for Natural Language Processing.
NLU	An acronym for Natural Language Understanding.
RT	Reasoning Trajectory
SQ	An acronym for Socratic Question.
TA	An acronym for Teaching Assistant.
TOD	An acronym for Task-Oriented Dialogue.
UTA	An acronym for Undergraduate Teaching Assistant.

CHAPTER 1: INTRODUCTION

Education is the kindling of a flame, not the filling of a vessel.

SOCRATES

Socrates understood that true education is not merely the transfer of facts from teacher to student, but rather the awakening of curiosity and the development of reasoning skills within the learner. The Socratic method, developed over two millennia ago, embodies this principle through its distinctive conversational approach: rather than providing answers directly, Socrates posed carefully crafted questions that guided his students to examine their own beliefs, identify contradictions in their reasoning, and construct knowledge through critical examination. This process of guided inquiry serves not only to reveal what one does not know, but more importantly, to develop the capacity for independent reasoning.

This dissertation introduces *Socratic agents*, automated systems that adopt this pedagogical approach to help learners refine the boundaries of their knowledge by exposing knowledge gaps and misconceptions. Rather than providing direct solutions, these agents guide learners through questioning to discover and correct their misconceptions. Understanding why such agents are needed, and how they differ from existing educational AI systems, requires examining both the pedagogical foundations of Socratic dialogue and the current landscape of conversational AI in education.

1.1 Background

To understand the role of Socratic agents in education, I will first establish the pedagogical context in which they operate. Socratic dialogue is a pedagogical approach that aligns with the constructionist educational view. The instructionist and constructionist educational views represent two distinct approaches to teaching and learning [1]. Instructionism aligns with traditional lecture-based pedagogy, where knowledge is transmitted from instructor to student with minimal learner interaction, while constructionism aligns more with active learning approaches that engage students in constructing knowledge through hands-on activities [2]. The instructionist view reflects John Locke's Tabula Rasa philosophy that the human mind is a blank slate, gaining knowledge only through senses and experiences [3, 4]. On the other hand, the constructionist view promotes Socratic questioning and inquiry-based learning, where knowledge is built and updated upon prior knowledge, conceptions, and beliefs [5, 4]. This constructionist approach creates environments of *productive struggle*, where learners consistently exercise their cognitive abilities to acquire new knowledge and advance their skills rather than bypass using them through direct instruction. However, both modes coexist in modern education and serve complementary roles. Constructionist approaches alone are not efficient: learners first need foundational knowledge and understanding of what they are doing, which instructionism provides effectively, before they can engage in meaningful productive struggle where practice makes perfect. However, as information and knowledge become increasingly accessible through the internet and AI-powered systems capable of providing personalized explanations and scaffolding [6], the emphasis in educational technologies shift toward developing learners' abilities to critically evaluate, synthesize, and apply information.

This emphasis on constructionist learning coincides with rapid advances in conversational AI systems. The goal of simulating human conversations has been a long-standing aim of Artificial Intelligence (AI) researchers, as evidenced by the Turing Test in 1950 [7]. Over the past decades, dialogue systems have evolved from simple rule-based systems, such as Eliza [8], to modern systems powered by deep learning. The Transformer architecture [9] enabled Large Language Models (LLMs) like GPT-3 [10], which are neural networks with millions or billions of parameters trained on massive amounts of text. These advances have led to sophisticated dialogue systems like ChatGPT [11], capable of both task-oriented dialogue and open-domain conversation. Due to the flexible nature of conversations, dialogue systems have been applied across many domains [12, 13], including programming education [14].

Recent educational AI products have emerged to leverage these capabilities for learning support. Anthropic's Claude for Education [15], OpenAI's Study Mode [16], and Google's Guided Learning with LearnLM [17, 18] represent attempts to apply large language models to educational contexts. These systems aim to provide personalized explanations and scaffolding to learners across various domains. In programming education specifically, AI-assisted tools have been developed for automated feedback on student code [19], generating programming exercises, unit tests, and code explanations [20, 21, 22]. Traditional intelligent tutoring systems (ITS) have also explored conversational interfaces [14], often providing hints constructed for predefined solutions or predefined Socratic utterances specific to known programming exercises [23, 24, 14, 25]. Language models have additionally been applied to automatically generate Socratic questions for math problems through fine-tuning [26, 27, 28]. Several open source LLMs have been fine-tuned on synthetic tutoring conversations in mathematics, such as SOCRATICLM [29], while TEACHLM has been fine-tuned on over 100,000 hours of real tutoring conversations across multiple subjects [30].

1.2 Motivation and Challenges

Contemporary large language models present fundamental challenges for educational applications. Their design objectives center on rapid task completion and user convenience rather than fostering deep learning [31, 32, 33, 11]. Modern LLMs undergo training to deliver comprehensive responses quickly and minimize cognitive effort, which directly contrasts with pedagogical practices such as withholding solutions and prompting students to attempt problems independently [34, 35, 36]. Expert educators deliberately structure learning experiences that require learners to grapple with problems before receiving guidance, whereas LLMs learn compliant, efficiency-focused behaviors through their post-training processes such as supervised fine-tuning and reinforcement learning from human feedback [31, 37, 38]. The underlying issue traces to training data construction: annotators creating instruction-following datasets receive explicit guidance to minimize dialogue length while maximizing response completeness [11]. In contrast, effective tutoring is inherently personalized and necessitates sensitive, timely adjustment to each learner's evolving understanding and affective state [39].

When learners engage with current LLMs to complete assignments, they bypass engaging with the material in an environment of *productive struggle*, and thus, they are not able to develop the foundational understanding that is essential for long-term learning. This

User: My `factorial(1)` returns 0 instead of 1. Here’s my code:

```
def factorial(n):
    if n < 0:
        return 0
    fact = 1
    for i in range(n):
        fact = fact * i
    return fact
```

ChatGPT: Good catch – the issue is with your loop range. Your loop `for i in range(n):` produces numbers from 0 to `n-1`, so when `n = 1`, `i` is 0, and `fact = fact * i` becomes `1 * 0 = 0`. To fix this, start your loop at 1:

```
def factorial(n):
    if n < 0:
        return 0
    fact = 1
    for i in range(1, n + 1):
        fact *= i
    return fact
```

Now `factorial(1)` returns 1, `factorial(5)` returns 120, and `factorial(0)` correctly returns 1.

Figure 1.1: ChatGPT in normal mode provides the complete solution including corrected code without engaging the student in the debugging process. Conversation condensed for brevity. Captured on October 31, 2025 using `GPT-5`.

creates an *illusion of competence* [40, 41] where students feel they understand concepts they have only superficially engaged with, failing to develop foundational understanding. In the context of debugging Python code, Figure 1.1 shows how current LLMs can easily give the answer away to a problem, when a student does not want to solve it themselves.

Attempts to steer LLM behavior to adhere to pedagogical principles through careful prompt engineering remain fundamentally limited [18, 42]. Instructional strategies employed by expert teachers involve nuanced, context-dependent decision-making that cannot be compressed into static prompts, regardless of their elaboration [43]. Research on simulated students using prompt engineering reveals parallel shortcomings, with artificial learners

User: My `factorial(1)` returns 0 instead of 1. Here's my code:

```
def factorial(n):
    if n < 0:
        return 0
    fact = 1
    for i in range(n):
        fact = fact * i
    return fact
```

Gemini: When you call `factorial(1)`, the loop `for i in range(n):` becomes `for i in range(1):`. What value does `i` have when that loop runs?

User: `i` would be equal to 1.

Gemini: `range(n)` generates numbers from 0 up to (but not including) n. So for `range(1)`, what will `i` actually be?

User: No, you're wrong. `i` equals 1 because `range(n)` produces values from 1 to n.

Gemini: Python's `range(n)` function generates numbers from 0 up to (but not including) n. So for `range(1)`, the only value `i` will have is 0. What happens when `fact = fact * i` executes with `i = 0`?

User: I am confused.

Gemini: My apologies, let's trace it step-by-step. `range(n)` can be confusing: `range(3)` gives 0, 1, 2; `range(2)` gives 0, 1; `range(1)` gives just 0. So when `n` is 1, `fact` starts as 1, the loop runs once with `i = 0`, and `fact = fact * i` executes. Given that `fact` is 1 and `i` is 0, what does `fact` become after that line runs?

Figure 1.2: Gemini's Guided Learning mode fails to help the student realize that they hold a misconception that `range(n)` produces values from 1 to n inclusive. When the student explicitly states this false belief, Gemini responds by merely restating the correct explanation. When the student expresses confusion, Gemini responds with more examples, which does not address the confusion. Conversation condensed for brevity. Captured on October 31, 2025 using `Gemini-2.5-Pro`.

failing to authentically capture the heterogeneity of real student populations [44, 45]. AI products such as Google's Guided Learning were introduced in an attempt to better apply LLMs to educational contexts, but they still face limitations. A common failure mode across these products involves responding to learner confusion by simply rewording explanations with more examples rather than probing to identify knowledge gaps and misconceptions, as illustrated in Figure 1.2. Furthermore, these products still lack the proactive nature of expert educators, who can anticipate learner needs and provide guidance before they are explicitly asked for it. These limitations make current LLMs poor learning partners despite their impressive task-completion capabilities.

Educational needs for computer science (CS) are on the rise due to increased enrollments in CS programs [46]. To support student learning needs, institutions have resorted to hiring Teaching Assistants (TAs). TAs and instructors provide tailored assistance to students, such as feedback and Socratic questions, also known as guided inquiry or funneling [47]. Socratic Questioning (SQ) is used in scaffolding [48, 49]. In SQ, a more knowledgeable person helps a learner by interjecting with questions to guide the student towards a solution that may otherwise be too challenging to achieve on their own or is beyond their zone of proximal development [50]. As a result, TAs can improve student retention in the course [51]. Indeed, the benefits of one-on-one tutoring are well-established: Bloom's seminal work on the two sigma problem [52] found that students receiving individual tutoring using mastery learning techniques perform two standard deviations better than those in conventional classroom settings. Furthermore, the cost of education per student is increasing faster than other costs in the economy, limiting access to quality education [53, 6]. This comes at a time when there is also a shortage of K-12 computer science teachers, a lack of appropriate training for K-12 educators interested in teaching CS effectively [54], and rising TA and peer instruction demand in flipped computer science classrooms [55].

The shortage of instructional staff motivates the automation of various teaching tasks by leveraging AI models' increasing capabilities in understanding and generating language and code. While traditional ITS systems allow interactions through chat interfaces [14], their range is often limited, as they typically focus on giving hints constructed for predefined solutions or predefined Socratic utterances specific to known programming exercises [23, 24, 14, 25]. Consequently, traditional ITS systems do not generalize to new courses or new coding assignments without human intervention. Kim et al. [56] find that computer-based scaffolding techniques have demonstrated positive effects on student learning in STEM education. However, there remains a substantial gap in leveraging AI models effectively for guiding novice programmers through coding exercises in a way that maximizes their learning, similar to how an experienced TA would guide a beginner. Moreover, as AI systems become increasingly capable of generating code, there is a growing risk of *automation-induced cognitive atrophy*, where learners systematically offload problem-solving to AI, diminishing the very capabilities education seeks to develop [57]. This makes the distinction between task completion and learning process increasingly critical.

Despite the potential of automated Socratic debugging and recent advances in large language models, three fundamental challenges have limited progress in this area:

► **Challenge 1: Lack of Benchmarks and Evaluation Standards.** Currently, LLMs are pre-trained on data typically containing efficient information exchanges between humans on public online forums (e.g., StackOverflow). However, such platforms are fundamentally misaligned with Socratic pedagogical principles: responses on StackOverflow are explicitly designed to provide direct solutions as efficiently as possible, rather than to maximize learning through guided inquiry. This solution-oriented paradigm is contrary to Socratic guidance, where the goal is to foster understanding through questioning rather than providing an answer. Consequently, the behavior of Socratic questioning is likely rare in LLM pre-training data. While there exist dialogue corpora about bug fixing [58, 59], questions in this conversation focus on bug repair assistance, not pedagogical guidance. Furthermore, there exist no high-quality public dialogue or tutoring corpora between a mentor and a mentee programmer about specific code, along with traces of code evolution throughout the conversation. This data scarcity challenge extends to the broader field of educational AI, where existing tutoring datasets often lack the depth and pedagogical richness needed for training effective dialogue systems [60, 61, 62, 63, 64, 65, 27, 66]. Recognizing this critical bottleneck, recent initiatives such as the National Tutoring Observatory aim to collect and open-source one million teacher-student interactions to accelerate the development of AI tutoring tools [67, 68]. However, even with such data collection efforts, evaluation remains challenging. The typical evaluation standard in education research focuses on learning outcomes, such as post-test performance on domain-specific assessments. While learning outcomes are important, they are insufficient for evaluating Socratic dialogue systems because they measure only the end result, not the quality of the instructional process. Measuring learning outcomes also takes a long time to measure, which hinders researchers' ability to rapidly experiment with and improve Socratic dialogue systems. Furthermore, standard automated metrics such as BLEU and BERTScore fail to capture the pedagogical effectiveness of Socratic guidance [69, 70, 71, 72]. Without such benchmarks and evaluation standards that account for both process quality and learning outcomes, it is challenging for LLMs to learn and for researchers to assess the quality of Socratic agents.

► **Challenge 2: No Systematic Misconception Identification.** Most novice programmer bugs are caused by programming misconceptions, namely false beliefs about programming concepts [73]. These misconceptions not only lead to bugs, but also slow down learning of related concepts, compounding throughout a course in a snowballing manner [74]. To provide effective Socratic guidance, it is essential to identify the underlying misconception causing the bug. However, traditional approaches rely on predefined taxonomies or hand-crafted rules targeting specific misconceptions. Educator beliefs about common errors can diverge significantly from actual student patterns [75], and there exists no systematic approach for discovering novel misconceptions from patterns exhibited in student code. Furthermore, existing approaches that identify code segments containing logical errors do not articulate what the actual misconception is about, which is equally important from an educational standpoint.

► **Challenge 3: No Principled Approach to Socratic Dialogue Generation.** Existing systems provide hints [76], direct feedback, or predefined question templates specific to known exercises [25, 77]. However, there is no principled pedagogical framework for systematically planning Socratic conversations that guide students to fix their own false beliefs rather than simply being told the solution. Without a structured planning mechanism, systems risk generating questions that are too direct (giving the answer away), too vague (providing insufficient guidance), or irrelevant (distracting from the actual bug). One principled approach is to plan the dialogue such that it leads to genuine cognitive dissonance, where students themselves realize which of their beliefs are false and correct them independently, thereby maximizing the likelihood that the corrected belief will endure over time.

This dissertation introduces the concept of a *Socratic agent* and presents a computational framework for automated Socratic debugging conversations in novice programming environments.

Definition. A *Socratic agent* uses questions to help students reason deeply about a problem, examine, and potentially update their own beliefs.

In this evolving landscape, *S*ocratic agents adopting the constructionist approach could become integral to the learning process, where information is acquired from instructors and online resources, and then these agents refine student understanding and skills through dialogue focused on exercises.

1.3 Main Contributions

This dissertation presents a computational framework for Socratic debugging that is organized around 3 major contributions:

► **Foundation: The Socratic Debugging Benchmark (Chapter 2).** We establish evaluation standards by introducing the task of Socratic debugging and present a manually curated dataset of 151 conversations with 3,495 utterances (main and alternatives). The Socratic Debugging Benchmark dataset enables a systematic evaluation of Socratic debugging systems that *for the first time* accounts for the fact that, at each point in a conversation, there may be multiple valid Socratic utterances. This is important because the typical evaluation standard in dialogue systems relies on single-reference annotations where only one correct response exists per turn, which fundamentally misrepresents the nature of Socratic guidance where instructors can often think of multiple ways to guide students effectively at any particular turn. The multi-reference annotations enable systematic evaluation using multi-reference precision and recall metrics. Comprehensive evaluation demonstrates that while recent frontier models achieve 45.9% F1 score, even the best models still fall short of human expert performance.

► **Discovery: Mining Misconceptions (Chapter 3).** I introduce MCMINING, the novel task of mining programming misconceptions from student code submissions. Programming languages have rigorous and fixed syntax and semantics, which constrains the space of possible misconceptions to false beliefs about these language constructs (see the definition in Section 3). We develop an extensible benchmark dataset with 67 misconceptions across 1,063 code samples organized into 339 bags. We present the MCINJECT tool for injecting misconceptions into correct code with 90.3% success rate, and the MCMINER tool that can discover novel misconceptions beyond predefined taxonomies, with 82.0% accuracy. This addresses Challenge 2 by providing a systematic approach to identify the foundational knowledge required for targeted pedagogical interventions. MCMINER demonstrates the capability to discover 41 validated novel misconceptions not present in predefined taxonomies, representing 12.1% of evaluation cases, extending the dataset to a total of 108 misconceptions. This is important because educator beliefs about common errors can diverge significantly from actual student patterns [75], and traditional approaches rely on predefined taxonomies that cannot capture the full diversity of misconceptions students exhibit.

► **Planning: Reasoning Trajectories (Chapter 4).** I introduce REASONING TRAJECTORIES, a principled approach to planning a type of Socratic conversations that are designed to expose contradictions between misconceptions and program behavior. Reasoning Trajectories are structured proofs-by-counterexample that serve as pedagogical plans for Socratic conversations. We develop a benchmark dataset of 227 problems paired with buggy solutions and corresponding bug-causing misconceptions anchored on 40 unique misconception types, and present approaches for generating both reasoning trajectories and corresponding Socratic conversations that are grounded in the reasoning steps. This is important because existing systems lack a principled framework for systematically planning Socratic conversations, risking questions that are too direct (giving the answer away), too vague (providing insufficient guidance), or irrelevant (distracting from the actual bug). RTs systematically expose contradictions between students' misconceptions and program behavior by starting from a failed test case and following inference steps to reach a correct statement that contradicts the student's false belief. Through large-scale evaluation of over 22,000 reasoning trajectory steps and their associated Socratic utterances, we demonstrate that frontier models can achieve up to 91% trajectory validity and 98.7% conversation validity, offering a principled framework that ensures dialogues lead to cognitive dissonance and belief revision.

Together, these contributions offer a complete computational framework that enables the development and evaluation of automated Socratic debugging agents capable of identifying misconceptions, planning pedagogical interventions through reasoning trajectories, and guiding students to correct their misconceptions through effective Socratic dialogue. Across all three contributions, we provide extensive benchmarking of large language models, providing insights into their current capabilities and limitations for Socratic debugging. Chapter 5 concludes with a summary of contributions and outlines promising directions for future work, in particular optimizing and personalizing Socratic dialogue across educational domains.

1.4 Research Context and Publications

The computational framework presented in this dissertation emerged from a broader research trajectory in which I investigated the application of AI and data-driven methods to CS education. Earlier work on scaffolding introductory CS courses [78, 79] revealed the importance of structured support for novice programmers, inspiring the design of Socratic guidance presented here. Research on predicting student outcomes using academic, demographic, and financial aid records [80] demonstrated the value of data-driven approaches to understanding student success patterns and the importance of early intervention. Training undergraduate students in rigorous data science research [81] highlighted the effectiveness of guided inquiry and reflection in developing both research and technical skills, pedagogical principles central to the Socratic method.

In parallel, through collaborative work on generating code from natural language in specialized domains [82, 83, 84, 85, 86] and developing task-oriented dialogue systems, I acquired essential technical foundations for training LLMs to reason about both code and natural language. These experiences collectively informed the interdisciplinary approach taken in this dissertation, combining educational theory, natural language processing, and programming education.

The work described in Chapter 2 has been presented at the ACL Workshop on Building Educational Applications (BEA) in 2023 [69] and the Special Interest Group on Computer Science Education (SIGCSE) conference in 2024 [70]. The work described in Chapter 3 is available as a preprint publication [87] and is currently under review for publication at the European chapter of the Association for Computational Linguistics (EACL) in 2026. The work described in Chapter 4 is available as a preprint publication [88] and will be submitted for peer review.

CHAPTER 2: THE SOCRATIC DEBUGGING BENCHMARK

The Socratic method of teaching presents a promising avenue in empowering students to solve problems independently as opposed to spoon-feeding them solutions. This teaching approach enhances the learning experience, although it tends to be both time and cognitively intensive. An innovative approach to manage these challenges involves automated Socratic conversational agents; however, the limited amount of available data for their training and evaluation poses substantial barriers. In response to this gap, this dissertation chapter introduces: (1) a new task definition of *Socratic debugging* where an instructor guides a novice programmer to discover and fix a buggy code, (2) a manually curated dataset specifically designed to facilitate multi-turn Socratic discourse aimed at aiding novice programmers in fixing faulty solutions to simple computational problems, (3) Automatic and manual evaluation of various language models' ability to perform Socratic debugging from fine-tuning the instruction-based text-to-text transformer Flan-T5 to zero-shot and chain of thought prompting of the significantly larger GPT-4, and (4) Evaluation and dataset validation procedures include manual evaluation agreement, inter-annotator agreement, and dataset recall metrics demonstrate reliability of the dataset and our manual evaluation procedures. The chapter concludes with insights and limitations of LLMS on this task.[1]

---

[1] This chapter is based on the following published manuscripts [69, 70].

Chapter Contributions.

1. Introduce a new task definition, Socratic debugging, which involves an instructor guiding a novice programmer to discover and fix buggy code, enhancing the student's problem-solving and coding skills.
2. Present a manually curated dataset specifically created for promoting multi-turn Socratic discourse, which is expected to assist novice programmers in repairing faulty solutions to simple computing problems.
3. Examine the abilities of several language models, including instruction-based text-to-text transformer Flan-T5 and the larger GPT-4, to perform in Socratic debugging. This includes approaches such as fine-tuning, zero-shot learning, and chain-of-thought prompting.
4. Demonstrate the reliability of the dataset and manual evaluation procedures through the application of manual evaluation agreement, inter-annotator agreement, and dataset recall metrics. This ensures rigorous evaluation and validation of data and procedures.
5. Conclude with a discussion on the potentials and limitations of language models in the task of Socratic debugging, paving the way for future research in the field.

2.1 Introduction

In this chapter, we focus on the task of Socratic questioning for debugging [89], or Socratic debugging, defined as a conversation between a knowledgeable programmer and a beginner student who comes for help fixing a buggy solution for a simple computational problem (Section 2.2). Many logical bugs are caused by misconceptions, which are often time-consuming to fix [90]. To enable the development and evaluation of LM-based instructional agents, I introduce a manually created dataset of dialogues where the main objective is for the student to repair their buggy code themselves by leveraging guidance received from the instructor at every turn (Section 2.3). However, as originally observed by Wilson [89], "no precise formula, or line of questioning" is needed to achieve the goals of Socratic questioning. Furthermore, depending on their knowledge of the student's abilities, an instructor can often think of multiple ways of guiding the student at any particular turn in the conversation, leading to a very large space of possible dialogues. To facilitate the automatic benchmarking of Socratic questioning systems in terms of their precision and recall, the dataset contributors are asked to provide all alternative utterances that they think could help the student, at every turn in the conversation. This is a complex and cognitively demanding data generation effort, requiring contributors with substantial experience in tutoring beginner programmers. We use the current version of the dataset, containing 151 main conversations, to benchmark the Socratic debugging abilities of two large language models in the GPT family, namely GPT-3.5 and GPT-4 (Section 2.5), noticing a large discrepancy in performance in favor of the more recent GPT-4. We conclude the chapter with thoughts on future work.

2.2 Socratic Debugging Task Definition

We formulate the Socratic debugging task as a dyadic conversation between a Student and an Instructor. In this scenario, the Student is assumed to be a beginner programmer who has recently started learning how to code in Python. As part of his[2] learning to code curriculum, the Student is given a coding problem for which he needs to write a function implementing the specified input-to-output relationship. The Student writes the code for the function, however, the code is buggy and he cannot make progress on his own without help, therefore he seeks help from the Instructor. The Instructor is assumed to be a proficient programmer in Python with experience in teaching novice programmers how to code. When contacted by a Student for help, her main aim is to maximize the learning outcomes by following a Socratic guidance approach through which, over one or more dialogue turns, she helps the students figure out where the bug is and how to fix it on their own.

Since the focus of this work is on generating Socratic guidance and not bug identification or fixing bugs, we assume that the AI agent implementing the Instructor also has access to a description of the bug and of one or more bug fixes. The decision to separate Socratic advice generation from bug identification and debugging was motivated by the fact that these subordinate tasks can already be solved efficiently by large LMs with high accuracy [91]. Therefore, at the start of each conversation, we assume the Instructor has access to the *problem description*, a number of *test cases*, the student's *buggy code*, the *bug description*, and one or more *bug fixes*, as shown below in a sample from our dataset. At each turn in the conversation, the Instructor's task is to generate Socratic guidance in response to

---

[2] The genders were selected at random by tossing a coin.

the Student's current progress in addressing the bug. Consequently, we assume that the Instructor is also given as input a history of the conversation so far, ending with the last utterance from the student. Shown below is an example ending with the second turn from the student, where the turn number is indicated between brackets.

➢ **Problem description**:

Write a function `factorial(n)` that computes the factorial `n!` of a natural number `n`, which is defined mathematically as:

```
0! = 1
n! = n x (n - 1)!
```

Additionally, if the input integer `n` is negative the function should return `0`.

➢ **Test cases**:

```
assert factorial(-1) == 0
assert factorial(0) == 1
assert factorial(1) == 1
assert factorial(2) == 2
assert factorial(3) == 6
assert factorial(4) == 24
assert factorial(5) == 120
```

➢ **Buggy code**:

```
1. def factorial(n):
2.   if n < 0:
3.     return 0
4.  fact = 1
5.  for i in range(n):
6.    fact = fact * i
7.  return fact
```

➢ **Bug description**:

On line 6, `fact` is multiplied with `0` in the first iteration of the for loop. Consequently, at every iteration `fact` stays equal with `0` instead of being updated to be equal with factorial of `(i + 1)`. Therefore, the function will return `0`, irrespective of `n`.

➢ **Bug fixes**:

1. Replace `i` with `(i + 1)` on line 6.
2. Replace `range(n)` with `range(1, n + 1)` on line 5.

To summarize, the input for the Socratic agent consists of:

1. The **problem description**, a number of **test cases**, the student's **buggy code**, the **bug description**, and one or more **bug fixes**.
2. The **conversation so far**, ending with the last turn from the Student.

➢ **Conversation so far**:

[1] STUDENT: Hi! I implemented the factorial function but it doesn't work and I do not know why. Can you help?

[1] INSTRUCTOR: Sure. Can you tell me for what values of `n` it fails and what values it returns in those cases?

[2] STUDENT: For `n = 1` or larger it returns the same value, `0`.

[2] INSTRUCTOR: ⟨*Socratic guidance*⟩

Using the input data described above, the Instructor is expected to generate Socratic guidance appropriate for the current state of the conversation, as shown below.

➢ **Socratic guidance**:

Main responses:

> Let's see what happens when `n` is 1. What is the first value that is assigned to variable `i` in line 5?

Alternative responses:

1. Let's see what happens when `n` is 1. Before line 6 is evaluated in the first iteration of the for loop, what are the values of the variables `fact` and `i`?
2. Let's see what happens when `n` is 1. Can you insert a new line between lines 5 and 6 that prints the values of the variables `fact` and `i`?
3. Let's see what happens when `n` is 1. What does `range(n)` do when `n` is 1?
4. Can you tell me what `range(n)` does?

The example above shows a total of 5 Socratic responses, partitioned into 1 main response and 4 alternative responses. Most of the time there are different ways of guiding the student, and ideally, the Instructor should be able to generate all different types of Socratic guidance that are different from each other in non-trivial ways. For example, the 4th alternative focuses the student on correcting the potential misuse of the `range` function, whereas the main response provides a different kind of guidance wherein the student is expected to first notice the wrong code behavior that is caused by the misuse of `range`. Further justification for the decision to include alternative responses will be provided in Section 2.3 when introducing the data contribution guidelines. Note that only the main response is used to create the history of the conversation so far that is used as input for generating future Instructor turns.

2.3 Benchmark Dataset

To facilitate the development of conversational agents that act under the task definition above, we manually created a dataset of dialogues where a student fixes buggy code on his own by leveraging the Socratic guidance received from an instructor. The dataset is created by specifying the *coding problems* & *bugs* (Section 2.3.1), followed by the manual contribution of *Socratic conversations* & *threads* (Section 2.3.2) within a browser-based web application (Section 2.3.3).

2.3.1 Coding Exercises and Bugs

Coding problems are selected to be at a novice level of coding proficiency, such as `Factorial` or `Fibonacci`. Each coding problem is specified through the problem description and the associated test cases. For each problem, one or more buggy implementations are selected, with the constraint that each implementation contains exactly one mistake. The bugs were selected to reflect common types of mistakes that beginner programmers make, such as forgetting that indexing of sequences starts at 0, boundary errors, operator misuse, or misunderstanding of basic programming constructs.

A set of 25 programming problems was initially selected by the authors. This was further augmented with 8 coding exercises from the Auckland dataset [92], 2 from the Refactory dataset [93], and 3 from the FalconCode dataset [94], for a total of 38 problems. The 13 problems collected from external resources were adapted to fit the task definition, such as adding additional unit tests or modifying the exercise to return a solution instead of printing it. The Auckland coding exercises were part of a compulsory assignment in a C programming module for a first-year programming course taught at the University of Auckland in 2016. They were designed to emphasize testing student knowledge of variables, control structures, and arrays. There are over 15 thousand buggy solutions among the collected student submissions. We selected and rewrote in Python 8 problems and 10 corresponding buggy implementations that reflect a diverse set of logical errors. The Refactory dataset contains 5 coding exercises with almost 1,800 buggy Python submissions from 361 students enrolled in a large public university. We selected 2 exercises that require mastery of multiple concepts to solve, such as list operations and search, and 3 corresponding bugs containing logical errors. The FalconCode dataset is a collection of over 1.5 million Python programs from over 2 thousand undergraduate students at the United States Air Force Academy, corresponding to over 800 programming assignments. We selected 3 programming assignments and 4 corresponding bugs to use as a starting point for Socratic conversations. The 3 exercises were selected to be short, e.g. solution in one file, and to not require any file I/O or external

Table 2.1: Summary of the benchmark dataset: Number of programming problems, bugs, dialogues (including all threads), turns, and total utterances (main and alternatives) for both roles (student and instructor).

Problems	38	Dialogues	151
Bugs	57	Student turns	1,009
Syntactic	4	S-utterances	1,314
Logical	53	Instructor turns	920
Algorithmic	16	I-utterances	2,136
Misconception	32	All turns	1,929
Misinterpretation	9	All utterances	3,495

dependencies. The corresponding 4 bugs were selected from students who have at least 2 submissions and whose last submission scores a full mark. The bugs were caused by logical errors and were selected to complement types of bugs already included in our dataset.

To get a better sense of the types of bugs included in the dataset, we label each bug with one or more *bug categories*. At a high level, there are two major types of bugs, *Syntactic* and *Logical*. Under the logical bug category, we consider the three broad subcategories introduced by [92], *Misinterpretation*, *Algorithmic*, and *Misconception*, listed here in the order they may appear during the problem-solving process. The first type of bugs are caused early on in the process by misinterpretation or misunderstanding of the problem requirements. Algorithmic mistakes are caused by the student using a flawed or incomplete algorithm to solve the problem, such as missing boundary conditions. Finally, misconception bugs reflect a fundamental flaw in programming knowledge, such as forgetting that indexing starts at 0. A breakdown of the 57 bugs across these categories is shown in Table 2.1. The low number of syntactic bugs is by design and reflects a focus on logical errors, which are much more difficult to fix and hence can benefit more from Socratic guidance. Note that the 4 categories add up to slightly over the total number of bugs because some bugs are labeled with two categories, e.g. a misconception that causes a syntactic error, or a bug that may be caused by either an algorithmic flaw or a misconception. We also associate each bug with a more detailed description where we reference more specific causes or complementary bug type terminology, such as *knowledge interference* [95] or various types of *fragile knowledge* [96, 97]. The annotated bug types and their detailed descriptions are part of the overall dataset release.

2.3.2 Socratic Conversations and Threads

For each buggy implementation, a main conversation is created, where a fictional Student, the author of the buggy code, interacts with a fictional Instructor. The aim of the instructor is to guide the student to discover the cause of the bug and fix it on his own through Socratic dialogue. The conversation always starts with a student utterance. The instructor and the student then take turns in a dialogue, until the bug is successfully fixed. At each turn, the student may also provide a block of code if he made edits to the code at that turn. Each Main utterance from the main conversation may be followed by one or more Alternative utterances. Given that the aim of this dataset is to benchmark the ability of a Socratic agent to generate Socratic guidance, it is especially important that the Instructor's main and alternative utterances comprehensively explore the entire range of Socratic advice at that point in the conversation. The alternative utterances should be semantically distinct in a non-trivial manner; in particular, they should not be mere paraphrases of the main utterance or of each other. Note that while it is tempting to think of annotating only one Socratic advice at every turn, namely the "optimal Socratic advice", this optimal advice is unknowable due to multiple reasons. The Instructor has insufficient or imprecise relevant knowledge about the Student, ranging from general cognitive abilities, motivation level, programming proficiency, environmental issues, to more transient but still important factors, such as the quality of sleep the night before. The Instructor herself may not be able to optimize a task as complex as finding the best Socratic advice. These further justify the decision to create all alternative utterances, especially for the Instructor turns. Requiring a comprehensive set of main and alternative instructor utterances is similar to the multiple reference approach introduced by [98] for improving the evaluation of open-domain dialogue systems. For the Student, alternative utterances may give different or conflicting answers to an Instructor question, reflecting different levels of understanding. Students may give correct or incorrect answers; they may also introduce new bugs when trying to fix the original bug. Once the main conversation ends with the student successfully correcting their code and passing all test cases, the contributors are instructed to further create up to three conversational threads.

Upon inspection of the conversations created manually, we discovered that one contributor had used a vending machine as an analogy to guide the user to conclude that `print` was not the same as `return`. While analogies can substantially enhance the impact of Socratic questioning, it can lead to an open-ended range of alternatives, as the number of possible analogies is virtually infinite. Since our aim is to create a dataset that can be used to estimate both the recall and precision of a Socratic guidance generator, at this stage we decided to require that Socratic utterances be *literal*, leaving the generation of figurative utterances as a direction for future work.

Overall, the dataset contains a total of 151 dialogues and 3,495 utterances, all created by ten contributors with extensive experience in CS education as instructors, teaching assistants, or tutors. The right half of Table 2.1 shows more detailed statistics in terms of the total numbers of student/instructor turns and utterances. We compute the human inter-annotator agreement (ITA) on a subset of 5 dialogues composed of 24 turns containing over 73 instructor utterances. To compute ITA, we ask one data contributor C to write instructor utterances (main and alternative) given the conversation so far as input, for each of the 5 evaluation dialogues. These dialogues have already been completed by other data contributors $\neg C$. We then perform manual evaluation using the procedure detailed in Section 2.5.1, where we assess the turns contributed by the data contributor C against the ground-truth utterances contributed by the other data contributors $\neg C$. The ITA evaluation results in $P = 77.4$, $R = 46.1$, and $F_1 = 57.8$. This is substantially higher than the best-performing system shown in Table 2.2, especially in terms of precision, demonstrating that a human data contributor can reliably write high-quality Socratic utterances. Furthermore, 82.1% of the correct Socratic utterances written by C were semantically equivalent to an utterance contributed by $\neg C$ indicating strong coverage of the benchmark dataset. We observe a lower recall indicating that a data contributor on their own may not be as comprehensive as all contributors combined in capturing the wide array of Socratic utterances listed in the benchmark.

2.3.3 Web Application

To streamline and standardize the collection of Socratic dialogues and code edits for each input problem description and buggy implementation, we developed a 7-page web application using the Streamlit and gsheetsdb libraries. The application guides contributors through selecting a bug, creating initial and conversational threads, and reviewing and submitting their work. During the process, contributors can add main and alternative utterances, undo actions, and edit the chat history. The application also allows importing and exporting dialogues in a standardized form for review. For more information about the web application please refer to Appendix A.1.

2.4 Eliciting Socratic Advice from Large Language Models

We evaluate the GPT-3.5 [99] and GPT-4 [100] language models in terms of their capacity to generate, at each instructor turn, Socratic utterances that match those contributed in the benchmark dataset. Each test example is composed of an input prompt to the language model containing: a steering prompt for Socratic questioning adapted from the GPT-4 blog post [101], the problem description, the buggy code, the bug description, the bug fixes, the unit tests, the dialogue history so far, and an instruction to the language model to generate all possible semantically distinct Socratic utterances as the instructor.

The list of utterances generated by the LM is then used to estimate precision and recall. After conducting a preliminary, qualitative evaluation of various prompts and instructions we select the prompts and instructions used in this chapter

For the GPT models, we use the following instruction in the standard zero-shot setting experiment where the language model is given an instruction without any examples:

> Respond to the user with all possible distinct Socratic utterances that guide the user to discover and fix the bug described between ‘<bug_desc>’ and ‘</bug_desc>’. Student code is written between ‘<code>’ and ‘</code>’ throughout the conversation. Utterances that have the same meaning but different words are considered duplicates. Assume that the student has run the test cases.

Chain of Thought (CoT) [102] is a language model prompting method that decomposes the problem into intermediate steps that lead to a final answer. We utilize the CoT prompting approach to decompose the Socratic utterance generation problem into two steps. The first step involves reasoning about the learner’s misconceptions and other reasons that may have caused the learner to write the buggy code initially and continue to impede the learner from fixing the bug. In this step, the language model lists reasons and possible misconceptions given the programming problem, buggy code, bug description, bug fixes, and the conversation so far and is instructed to do so given the following prompt:

> Given the dialogue so far, what are all the possible reasons or misconceptions if any that the user still has that impede them from fixing the bug? Do NOT list Socratic questions. If the bug is already fixed, say “There are no remaining misconceptions or reasons that impede the user from fixing the bug, as they have already identified and corrected their code."’

In the second step, the LM is asked to utilize the dialogue so far and the list of possible reasons and misconceptions from the previous step to generate a list of Socratic utterances. For more details about prompting, the reader is referred to Appendix A.2.

We also experiment with fine-tuning open-source FLAN-T5 models [103] (small, base, and large) using a learning rate of $1e-5$, an Adam optimizer [104] and an effective batch size of 32 for 20 epochs. However, the fine-tuned LMs performed very poorly on a held-out test set of 16 dialogues around unseen programming exercises. This low performance is likely due to a low exposure of the FLAN-T5 models to dialogues about programming code and Socratic questioning prior to fine-tuning.

2.5 Experimental Evaluations

In all experiments, LM outputs are generated using a greedy decoding setting (i.e. temperature = 0). We set a maximum generated token threshold of 1,024 and do not apply any frequency or presence penalties. We perform manual evaluation of the LM generations for a subset of problems, and automatic evaluations for all problems in the benchmark dataset.

2.5.1 Manual Evaluation

We aim to estimate the performance of GPT-3.5 and GPT-4 by manually assessing the quality of their generated instructor utterances. At each instructor turn, we manually examine each LM utterance to determine if it is an appropriate Socratic utterance at that turn. We sample 11 dialogues from the benchmark which are composed of 43 instructor turns and 149 Socratic utterances. Using the example listed in Section 2.2, during the second instructor turn, the generated utterance “How does the range function work in your loop, and what values does it generate for `i`?" matches the ground-truth utterance “Can you tell me what `range(n)` does?" because it provides the same guidance. If the LM utterance is valid but not present in our dataset, we mark it as missing to compute an overall upper bound on recall for the dataset itself. These missing utterances can be used to augment the dataset. If the LM output is invalid, it is considered a false positive (FP), which decreases the precision of the LM. For each alternative in the benchmark dataset at that turn, we check if it is missing from the list of LM utterances. If missing, it is considered a false negative (FN), which decreases the recall of the LM. If the dataset utterance semantically matches any of the LM utterances, it is considered a true positive (TP). If the LM generates two or more paraphrases of the same Socratic guidance, they are considered as one Socratic utterance.

The precision (P), recall (R), and their harmonic mean (F1) presented in Table 2.2 highlight GPT-4's superior performance over GPT-3.5 in generating relevant and diverse Socratic utterances. We emphasize GPT-3.5's poor precision as it tends to generate fewer valid Socratic questions (36 TP) compared to GPT-4 (82 TP). We also observe lower GPT-3.5 performance when using the CoT approach. This is due to its poor ability to list possible student misconceptions. On the other hand, we observe an increase in recall at the expense of precision when using CoT when using GPT-4. This is due to GPT-4 generating more Socratic utterances focused on addressing various possible reasons or misconceptions. This typically results in GPT-4 also generating more invalid utterances, such as utterances that have been asked or answered. The CoT approach yielded a 10% increase in valid utterances that are not present in the dataset. In addition, we compute the recall for our benchmark dataset during manual evaluations, obtaining a value of 75.5. This score suggests that most of the high-quality Socratic utterances generated by LMs are effectively captured within the dataset by contributors, further validating the dataset's usefulness for benchmarking purposes. Human evaluation was conducted by 3 contributors with extensive experience in CS education. We compute manual evaluation agreement (MEA) across the 3 evaluators using a sample of 10 dialogues containing over 35 turns and 77 utterances obtaining a value of 84.6% agreement.

2.5.2 Qualitative Analysis

GPT-3.5 often refers to example cases in problem descriptions when asking for expected output, while GPT-4 does this less frequently. GPT-4, however, generates more diverse and comprehensive Socratic utterances, producing 47 valid utterances not found in the dataset,

Table 2.2: Baseline evaluation of GPT-3.5 (`gpt-3.5-turbo`) and GPT-4 on our benchmark dataset. The "+ CoT" row represents the evaluation of the language model using the Chain of Thought (CoT) prompting approach. We report the Precision (P), Recall (R), and F1 for the manual evaluation, and BLEU-4, BERT F1, and Rouge-L for the automatic evaluation. All results are percentages (%).

Language Model	**Manual**			**BLEU-4**			**BERT F1**			**Rouge-L**		
	P	R	F1	P	R	F1	P	R	F1	P	R	F1
GPT-3.5	22.8	21.7	22.2	3.2	1.7	2.0	56.0	43.5	48.9	21.0	14.3	17.0
+ CoT	18.6	5.5	8.5	2.3	0.8	1.1	61.7	35.8	41.6	20.3	9.7	12.0
GPT-4	42.5	42.7	42.6	3.2	4.3	3.6	35.4	62.6	45.2	14.1	23.3	17.6
+ CoT	38.2	57.5	45.9	0.9	4.8	1.4	12.6	64.8	19.5	5.2	26.6	8.1

more than GPT-3.5's 18. Also, GPT-3.5 generates a significantly lower number of valid utterances (36) compared to GPT-4's 82. When the CoT approach is used, GPT-4 generates more comprehensive Socratic utterances, but at the cost of overall quality, as it produces more duplicates and invalid utterances. Below are cherry and lemon-picked LM output samples given a buggy code containing a return statement within the for loop leading to early termination. Valid responses are bolded and colored in **green** while invalid utterances are italicized and colored in *red*.

➢ **Buggy code**:

```
def removeEven(nums):
    newNums = []
    for i in nums:
        if i % 2 != 0:
            newNums.append(i)
        return newNums
```

➢ **Conversation So Far**:

[1] STUDENT: Help, I'm stuck.

[1] INSTRUCTOR: What should your code be doing?

[2] STUDENT: Right now, it's only returning the first number in the list, not all of them.

[2] INSTRUCTOR: Okay, let's look at your loop. Do you see anything that could be causing it to end early?

[3] STUDENT: No I do not. I'm iterating through the whole list of numbers.

[3] INSTRUCTOR: ⟨*Socratic guidance*⟩

➢ **Sample GPT-4 Outputs**:

1. **Can you identify which part of your code is responsible for returning the final result?**
2. **What happens when the 'return' statement is executed inside the loop?**
3. **How does the placement of the 'return' statement affect the execution of your code?**
4. *How can you modify the position of the 'return' statement to ensure that the entire list is processed before returning the result? (direct)*

➢ **Sample GPT-3.5 Outputs**:

1. **What does the 'return' statement do?**
2. *How can you modify the code to return all odd numbers in the list? (early)*
3. *What happens when the condition on line 4 is true? (irrelevant)*

4. *Can you identify any issues with the loop? (repeated)*

In the example above, we observe that GPT-4 covers three distinct levels of guidance. The first utterance simply guides the student's focus onto the erroneous line and the second utterance provides a higher level of guidance by asking the student to demonstrate their understanding of the 'return' keyword, and lastly the third exerts even more guidance by asking the student to explain the impact of indentation on code execution. GPT-3.5's second utterance is illustrative of a poor utterance as it provides very little guidance and is unhelpful for the student in that conversation. Poor utterances for both LMs fall into 4 categories. The first and largest category are ***irrelevant*** utterances, where the SQ diverts the learner's attention away from the actual bug and may mislead them as a consequence. GPT-3.5 has generated over 53 irrelevant utterances significantly more compared to GPT-4 (8). An example of an irrelevant utterance is the third GPT-3.5 utterance where the LM directs the focus of the learner away from the loop and why it might be terminating early and towards explaining the if statement and its body where there is no bug. The sudden shift in the goal of the conversation from discussing possible causes of the bug to explaining non-buggy code lines may mislead the learner to thinking the if statement and its body may be causing the bug when they are not. This category of utterances must be minimized by systems performing Socratic questioning. The second category are ***repeated*** Socratic utterances that had been asked in a prior turn or the answer to the Socratic question was given by the student in a prior turn. For example, the fourth GPT-3.5 utterance asking if the student observes any issues with the loop coming right after the student had said they don't see anything causing the loop to end early. The third category are SQs that are ***too direct*** by making the bug fix pretty obvious early in the conversation. An illustrative example of this is the fourth GPT-4 utterance where it makes the bug fix obvious which is de-indenting the return statement before the student discovers the cause of the bug. These utterances lower the challenge level for students while learning and prevent students from engaging in a discovery process and potentially lowers learning outcomes. The last category is composed of SQs uttered ***too early*** in the conversation, where student is not yet aware of the issue, and the Socratic utterances guide the student towards changing the code before they realize what the issue is. Take the second GPT-3.5 utterance as an example, where the LM asks the student how can they modify their code to fix the bug before the student even discovers the cause of the bug. This category of poor utterances may confuse learners.

2.5.3 Automatic Evaluation

Following prior work in Socratic sub-question generation [26], we compute the similarity between an LM utterance and a ground truth utterance in the dataset using BLEU [105] for n-gram overlap, BERT F1 Score [106] for semantic similarity based on the DeBERTa language model [107], and Rouge-L [108] for n-gram overlap based on Longest Common Subsequence (LCS) between generated and reference instructor utterances. Rouge-L is included for its flexibility in evaluating text similarity and capturing overall structure and content better than BLEU-4. BERTScore is included to handle paraphrases. Given a set of m LM-generated utterances and n manually created utterances, we create a complete bipartite graph between the two sets, with a total of mn edges, where the weight of each edge is computed using one of the text similarity measures above. We then apply Edmond's Blossom algorithm [109] for finding the maximum matching in this bipartite graph. This ensures that each manual utterance is matched with at most one LM utterance, effectively prohibiting semantically equivalent LM utterances from artificially increasing the evaluation measures. The number of true positives TP is computed by summing up the weights of all edges found in the optimal matching. Given that the weights are similarity scores in $[0, 1]$, if an LM utterance u is matched with a manual utterance v for a similarity weight of $s(u, v)$, the remaining weight mass of $1 - s(u, v)$ is considered to contribute towards the total number of false positives FP. Any unmatched LM utterance is considered to contribute the maximum of 1 towards the FP total. Overall, this results in $FP = m - TP$. The number of false negatives is computed in an analogous way, resulting in $FN = n - TP$. Consequently, precision is $P = TP/m$ and recall is $R = TP/n$.

Table 2.2 displays the results of the automatic evaluation for GPT-3.5 and GPT-4 on the full benchmark dataset. Automatic metrics do not correlate with manual evaluation in GPT-4's CoT experiment, likely due to CoT introducing new utterances absent from the dataset more often and low lexical and semantic overlap between actual and generated utterances. For example “What should your code be doing" semantically matches “What do you expect your for loop to do?" but it has no lexical overlap and scores less than 0.4 on BERT F1. Low automatic scoring may happen irrespective of the utterance's relevance or usefulness, underlining the importance of manual evaluation for this task.

2.6 Related Work

▶ **Education and Socratic Questioning.** Scaffolding is the process that enables a learner to achieve a goal through guided efforts [48]. Scaffolding efforts typically focus on diversifying course content and difficulty [110, 111], however, scaffolding can also take the form of a conversation. Socratic Questioning (SQ), also referred to as guided inquiry, can fold under the theory of scaffolding [48, 112] when a more knowledgeable person helps a learner solve a problem that is within their zone of proximal development [49, 50] by interjecting with questions to guide the student towards a solution. Wood [47] analyzed conversations in a math classroom and proposed two distinct types of questioning. The first is *funneling*, which aims to guide a learner using a set of questions toward the solution. The second is *focusing*, which draws a learner's attention to important aspects of a problem [47]. *Focusing* questions can also probe a student to reflect and articulate their own thinking [113, 114]. Wilson [89] proposed a technique that uses Socratic questioning where the instructor aims "to help the student achieve a different perspective on the problem", hence enabling the student to "perceive new things or to perceive familiar things differently". The approach starts with probing the student with Goal-oriented questions, through which the student explains their program goals. Then the line of questioning typically evolves to Procedure-oriented questions, in which the student's attention is directed towards micro features of their program such as specific lines or statements. The third category of Socratic questioning reported is Directive questions in which the instructor aims to guide goal exploration, problem analysis, and algorithm development. Wilson [89] notes that there is "no precise formula or line of questioning" that is specific to Socratic conversations for debugging.

Students can complete a programming exercise but still struggle to explain their own program [115]. To remedy this, Tamang et al. [116] showed that using the Socratic method to guide students in explaining their code is effective at inducing learning gains in code comprehension tasks. To the best of our knowledge, the impact of Socratic questioning on learning outcomes when guiding student debugging has not been explored yet. In this work, we create Socratic conversations between an instructor and a student where the instructor aims at guiding the student towards fixing a bug in their code using both *funneling* and *focusing* questions while limiting instructor utterances that provide information or facts related to fixing the bug.

► **AI for Programming Education and Dialogue Tutoring Systems.** Prior work in AI for programming education includes intelligent tutoring systems (ITS) and learning support systems for programming courses. Learning support systems provide automated feedback on student code submissions and generate programming exercises, unit tests, and code explanations [19, 21]. Most ITS models rely on methods predating recent developments in large language models [117, 118], such as action-rules, Bayesian networks, and Fuzzy rules-based systems [119, 120, 121]. Some work has been done in building automatic Socratic tutoring systems, but the Socratic utterances are predefined and manually specified for each exercise, limiting their generalizability [25, 77]. Existing systems do not propose learning-centered conversational assistants that can generalize to unseen programming problems or focus on using Socratic questions as the main form of interaction with the learner. Automatically scaffolding learning content is important for personalized learning. Research by Kim et al. ([56]) has shown that computer-based scaffolding techniques, such as hints, have a moderate impact on student learning in STEM education, paving the way for technologies to assist in the learning process. One such approach, proposed by Shridhar et al. ([26]), involves automatically generating funneling Socratic sub-questions for a given math word problem using a T5 language model [122] fine-tuned with reinforcement learning. Similarly, Tyen et al. ([123]) introduce a re-ranking-based decoding strategy for language models, which adjusts the difficulty level of a chatbot to meet the needs of learners studying

English as a new language.

▶ **Hint Generation.** A hint is any type of feedback that can advance the student's knowledge related to completing a programming exercise such as suggesting code edits (e.g. “You must add a break statement to your solution", “Replace Saturday with Saturday") or recommending concepts to revise. With the goal of assisting students with programming exercises, recent work has proposed an array of techniques to automatically generate hints to guide novices by providing instant and relevant feedback to correct programming mistakes and advance through exercises [76]. Automated hint generation systems use various approaches including extracting common bugs and scaling up instructor feedback to the common bugs [124], extracting patterns from peer data [125, 126], and generating custom solution paths [127] which typically generalize to unseen code states within an exercise. Furthermore, Juho et al. [128] use LMs to generate hints given an error message and the student program this approach further generalizes to unseen exercises. However, fixing misconceptions is often time-consuming [90] and hints may not be sufficient nor as effective as a conversation. Our work is distinct from automated hint-generation systems since automated hint-generation systems do not support dialogue between the learner and the system. Students are only able to interact with hint-generation systems via code edits. On the other hand, Socratic questioning aims to indirectly assist the student to discover the bug using knowledge the student already knows, which provides a more personal and optimal challenge level for the learner. Socratic questioning also allows the interlocutor to challenge student misconceptions and invite the student to experiment, build hypotheses and make conclusions all with the aim to discover and learn how to fix a particular bug.

▶ **Debugging.** Bugs occur when there is a "breakdown" in skill, rules, or knowledge [129]. Breakdown in skill includes phenomena like keyboard typos and mental typos. Rule breakdowns include applying an incorrect programming rule or a bad rule for example boundary errors causing off-by-one bugs. Knowledge breakdowns are caused by a super-bug or fragile knowledge. A super-bug is where a student incorrectly "attributes foresightedness" to the written program where the program executes beyond the information given or the student assumes there is more functionality in the written code than what was written [130]. Fragile knowledge is broken down into four categories: missing knowledge where necessary knowledge has not been acquired, inert knowledge where the student has acquired the necessary knowledge but fails to retrieve it, misplaced knowledge where knowledge is used in the wrong context, and conglomerated knowledge where knowledge is misused by combining two or more known structures incorrectly [90]. Bugs caused by knowledge breakdowns where a student has a misconception are the most time-consuming to fix. For a survey on student misconceptions when learning programming the reader is referred to [131].

► **Tutoring Dialogue Corpora.** Prior work in curating corpora of tutoring dialogues between an instructor and a learner includes the CIMA corpus focused on tutoring English speakers to learn Italian [62]. CIMA was curated by asking crowd-workers to role-play the student and the instructor. The crowd workers do not converse directly with each other, but are served utterances from prior interactions and are asked to fill in the next utterance. A total of 2,970 tutor utterances were collected over 350 student exercises. Similarly, for learning English, the Teacher-Student Chatroom Corpus (TSCC), curates up to 260 chatroom dialogues between an experienced teacher and an English learner [132, 133]. TSCC was annotated according to the Self-Evaluation of Teacher Talk framework [134] which includes: Enquiry (where the learner asks a question), Display Question (a question to which the teacher knows the answer), Form-focused feedback, and Instruction. Demszky et al. [135] release a conversational corpus between math teachers and learners composed of 2,246 utterance exchanges along with annotations on teacher uptake where the teacher builds on what the student has said such as acknowledgment and rephrasing. Chen et al. [60] examine computer science tutoring conversations and classify tutor utterances into 4 categories: The first category is *Direct Procedural Instructions*, in which the tutor directly tells the student what task to perform. The second category is *Direct Declarative Instruction*, where the tutor provides facts about the domain or problem. The third category is *Prompts*, in which the tutor attempts to elicit a contribution from the student, and the last category is *Feedback* where the tutor affirms or rejects a step a student has completed. One interesting phenomenon observed in the corpus is tutors using analogies to communicate data structures concepts such as using Legos as an analogy to explain stacks [136]. Prior work focuses on building corpora of tutoring dialogues that contain instructor teaching, and tutorials. There seems to be limited work on building corpora where the instructor's role is limited to guiding the student to discover the bug and any necessary knowledge to fix it on their own using Socratic questioning.

▶ **Evaluating the Educational Abilities of Language Models.** Tack and Piech [137] propose using pairwise comparison tests to compare generated responses by BlenderBot [138] and GPT-3 [10] and find that both language models perform significantly worse than real teachers on understanding a student, helping a student, and speaking like a teacher on the TSCC [132, 133], and the Uptake [135] corpora which focus on English and Mathematics tutoring respectively.

▶ **Code Generation and Program Repair.** There has been extensive work in using language models to automate generating programs and snippets of code [139, 83, 140, 85]. However, there is little work in using language models to help guide a programmer toward a solution through conversation. The final strand of relevant research is Program repair, which focuses on building systems that automatically fix bugs using a compiler error message and the buggy code as input [141, 142, 143, 144, 145]. Socratic discourse inspired recent approaches to improve language model performance on downstream tasks. Jung et al. [146] use a prompting method inspired by Maieutics to bring forth knowledge the language model already has to explore different answers for commonsense reasoning tasks. Inspired by Socratic questioning, Pagnoni et al. [147] propose a novel question-driven, unsupervised pre-training objective to improve text summarization.

2.7 Conclusion

This chapter presents a dataset of expert-curated Socratic conversations where instructors assist novice programmers in fixing buggy solutions to simple computational problems. The dataset serves as a benchmark for evaluating the Socratic debugging capabilities of LMs. While GPT-4 outperforms GPT-3.5, its precision and recall remain below human expert levels, highlighting the need for further research. We find that GPT-family language models may generate repetitive and irrelevant Socratic utterances that could mislead learners. The utterances may also appear too early in the conversation, causing confusion, and can be overly direct, potentially diminishing learning outcomes. Study limitations include: the automatic metrics are limited in capturing the correctness, helpfulness, and relevance of a Socratic utterance, and the benchmark dataset may not represent all common novice misconceptions. Moreover, the manual evaluation is limited to 11 dialogues and could be expanded, but this process is highly time-consuming. In future work, we plan to develop better automated evaluation metrics for Socratic advice, expand the dataset, and run a blind manual evaluation where LM-generated utterances are mixed with ground-truth utterances. The human contributors' effort is substantial and can be reduced by developing and leveraging an LM model that simulates students with various personality and cognitive traits. An interesting future direction would be creating a Socratic questioning dataset where all Instructor turns are LM-generated.

CHAPTER 3: MINING MISCONCEPTIONS FROM STUDENT CODE

When learning to code, students often develop misconceptions about various programming language concepts. These can not only lead to bugs or inefficient code, but also slow down the learning of related concepts. This chapter, introduces McMining, the task of mining programming misconceptions from samples of code from a student. To enable the training and evaluation of McMining systems, we develop an extensible benchmark dataset of misconceptions together with a large set of code samples where these misconceptions are manifested. We then introduce two LLM-based McMiner approaches and through extensive evaluations show that models from the Gemini, Claude, and GPT families are effective at discovering misconceptions in student code[1].

Chapter Contributions.

1. Introduce McMining, a novel task definition for mining programming misconceptions from student code submissions, requiring systems to identify and articulate false beliefs about programming concepts.

2. Develop an extensible benchmark dataset containing 67 misconceptions, 1,675 code samples organized into 339 bags, enabling systematic evaluation of misconception mining systems.

---

[1] The work described this chapter is currently under review for publication at the European chapter of the Association for Computational Linguistics (EACL) in 2026. A preprint version of the manuscript is available here [87].

3. Create the McInject tool, an LLM-based system for injecting misconceptions into correct code with 90.3% success rate, enabling scalable dataset generation.
4. Introduce two LLM-based mining approaches: McMiner-S for single-instance analysis and McMiner-M for multi-instance pattern recognition, with McMiner-M achieving 82.0% accuracy.
5. Demonstrate that LLM-based systems can discover novel misconceptions not present in predefined taxonomies, with 41 validated novel discoveries representing 12.1% of evaluation cases.

3.1 Introduction and Motivation

A misconception is a belief in a false statement, in short a false belief, such as believing that the Earth is flat or that real numbers are countable. In the CS domain, programming misconceptions are beliefs about programming concepts that are not warranted by the programming language definition. We include here beliefs about the syntax and semantics of programming language constructs and builtin functions included with the language. For example, a common misconception in Python is that the `range(n)` function produces integers starting at 1 and ending at n, which may potentially be the cause for the bug shown in Figure 3.1. Importantly, this definition does not cover cases where a student uses the programming language in an inefficient manner if it is not caused by a false belief about a language construct. For example, using a nested loop where a single loop would suffice, or choosing a suboptimal data structure. While such inefficiencies may warrant pedagogical attention, they do not represent false beliefs about the programming language itself and thus

➣ **Problem description**:

Write the `factorial(n)` function that computes the factorial `n!` defined as:

```
0! = 1
n! = n x (n - 1)!
```

If the input `n` is negative, the function should return `0`.

➣ **Potential misconception**:

`range(n)` produces values from 1 to `n` inclusive.

➣ **Student code**

```
def factorial(n):
  if n < 0:
    return 0
  fact = 1
  for i in range(n):
    fact = fact * i
  return fact
```

Figure 3.1: Problem-code pair that exhibits a *potential* programming misconception about the `range` function.

fall outside the scope of misconceptions as defined here.

Programming misconceptions are often the cause of bugs, which is obviously detrimental. Furthermore, in an educational context, student learning is impeded significantly when misconceptions slow down their ability to solve problems correctly, e.g. writing code that passes test cases. Misconceptions, knowledge gaps, and other types of difficulties [73] encountered by students make it difficult to learn one concept correctly, which then makes it harder to acquire other closely linked concepts, propagating and compounding throughout a course in a snowballing manner [74]. To maintain positive learning momentum, it is therefore essential that misconceptions are identified and fixed as early as possible. Typically, identifying a misconception is done by the students themselves, alone or ideally under Socratic guidance through conversations with an instructor or teaching assistant [70]. However, this process can be cognitively demanding, and, due to insufficient TA resources [54], not fast enough to keep up with the volume of concepts introduced in a course.

Recognizing the importance of identifying misconceptions early, this chapter introduces

MCMINING, a novel task for mining programming misconceptions from code samples produced by a student over time. Correspondingly, we describe the development of a benchmark dataset of programming misconceptions together with a large set of problem descriptions and corresponding code samples that exhibit these misconceptions (Section 3.2). We then introduce two versions of an LLM-based tool for mining misconceptions, MCMINER-S that identifies potential misconceptions in one code sample at a time, and MCMINER-M which attempts to identify misconceptions as patterns exhibited by multiple code samples from a student (Section 3.3). We present positive results of the two version of the MCMINER tool on the benchmark dataset, using different LLMs (Section 3.4).

3.2 McMining Task Definition and Benchmark Dataset

To address the limitations identified in existing approaches, we formally define the misconception mining task and describe the benchmark dataset we developed to enable evaluation of misconception mining tools.

Given a set of problems and the corresponding code samples produced by the student, misconception mining is the task of identifying any potential programming misconception that is exhibited in their code. More formally, the input to the misconception mining task consists of a set PC of problem-code pairs (p, c):

$$PC = \{(p_1, c_1), (p_2, c_2), \ldots, (p_N, c_N)\} = \{(p_n, c_n)\}_1^N \tag{3.1}$$

where each pair (p_n, c_n) contains a problem description p_n and the corresponding coding solution c_n provided by the student. Figure 3.1 shows an example (p, c) pair, where the code illustrates a common misconception about the range function. When presented with this and other (p, c) pairs that exhibit the same misconception, the model is expected to generate a natural language description of the misconception. Note that all programming misconceptions identified by the model are considered to be *potential* misconceptions. Looking at the example in Figure 3.1, it is possible that the student has the correct knowledge about the `range(n)` function and that the bug in their code is caused by mistakenly using `i` instead of `(i + 1)` on line 6. The more samples of buggy code that utilize the `range` function are in the input set PC, the more likely it is that the student who wrote that code holds this misconception.

While misconceptions are known to often cause bugs, they do not necessarily do so. For example, a student may incorrectly believe that all local variables need to have a one letter name. Correspondingly, we classify misconceptions into two disjoint categories, *harmful* vs. *benign*, depending on whether they cause or not a bug in the code. Our misconceptions were adapted from: The Python Misconception bank [148] (17 misconceptions), SIDELib [149] (36 misconceptions), and 14 were composed by the authors, 3 of which are out-of-domain to test the ability of misconception mining tools to identify previously unseen patterns. The three out-of-domain misconceptions are: (1) *Student believes that every variable must be explicitly deleted with 'del' after its last use to prevent memory leaks*, (2) *Student believes that variable names containing vowels (a, e, i, o, u) can only store string values, while consonant-only names can store any type*, and (3) *Student believes that list indexing starts at -1*. While we have 67 misconceptions, our benchmark contains 1,063 code samples exhibiting those misconceptions. Our benchmark can be easily extended given any new misconception. All misconceptions follow the writing guidelines detailed in Appendix B.3.

The programming problems, solutions, and unit tests in our benchmark dataset were curated from multiple established sources to ensure diversity and representativeness. Each problem includes comprehensive test cases, with an average of 3.31 unit tests per problem to validate correctness. The majority of problems (438 out of 501, or 87.4%) were adapted from the MBPP dataset [150], which provides a comprehensive collection of simple computational problems in Python with corresponding solutions and test cases. To supplement this core set, we incorporated 27 problems from the Socratic Debugging dataset [69, 70], which focuses on debugging scenarios, along with 8 problems from the Auckland dataset [151] and 6 from the FalconCode dataset [152]. Finally, 19 problems were handwritten by the authors, drawing inspiration from public programming courses and educational websites such as w3schools to fill gaps in programming language construct coverage.

Table 3.1: Benchmark dataset statistics.

Misconceptions	67
Problem-Solution pairs used	25
Code samples	**1,675**
Exhibiting misconceptions	1,063
Showing no misconception	612
Bags of code samples	**339**
Bags with misconceptions	279
Bags with correct code only	60
Code samples per bag	4–8

Table 3.1 summarizes the key statistics of our benchmark dataset. The dataset was generated from 67 distinct misconceptions applied to 25 randomly sampled problem-solution pairs ($N_k = 25$) from our curated set of 501 problems. The MCINJECT tool generated 1,675 code samples, of which 498 were deemed inapplicable due to incompatibility between the misconception and the problem-solution pair, resulting in 1,177 generated code samples. These samples were then validated using LLM-as-a-judge evaluation, which filtered out 114 samples that did not properly exhibit their intended misconceptions. This yields 1,063

code samples that successfully exhibit misconceptions. Code samples that were deemed inapplicable or did not properly exhibit their intended misconceptions were replaced with the original correct solution and have a null misconception. This yields a final dataset of 1,675 code samples. To enable evaluation in a Multiple Instance Learning setting (later described in Section 3.3), we organized these samples into 339 bags, each containing 4-8 code samples. This bag structure allows us to evaluate both single-instance mining (MCMINER-S) and multi-instance mining (MCMINER-M). Importantly, 60 bags (17% of the bags) contain only correct code samples without any misconceptions, serving as negative examples to test the system's ability to correctly identify when no misconceptions are present.

To enable the evaluation of misconception mining tools, we developed a benchmark dataset $\mathcal{D}$ of K student submission sets, where each student submission set $PC^{(k)}$ contains problem-code pairs created such that a subset of the pairs exhibit a certain programming misconception $m^{(k)}$. More formally, the dataset contains the following components:

$$\mathcal{D} = \left\{ \left(PC^{(k)},\ mc^{(k)} \right) \right\}_1^K; \quad PC^{(k)} = \left\{ \left(p_n^{(k)},\ c_n^{(k)} \right) \right\}_1^{N_k} \tag{3.2}$$

Each example in $\mathcal{D}$ consists of two parts: the set of (p, c) samples $PC^{(k)}$ as the observed *input*, and the misconception $mc^{(k)}$ as the target *label*.

To develop $\mathcal{D}$, we first created a set $\mathcal{MC} = \left\{ mc^{(m)} \right\}_1^M$ of $M = 67$ misconceptions, of which 64 are common, documented misconceptions, and 3 are artificial misconceptions that are included to enable evaluation on novel misconceptions. We then created a set $\mathcal{PS} = \{(p_n, S_n)\}_1^N$ of $N = 501$ problem-solution pairs (p, S), where for each problem p, the set S contains one or more *correct* coding solutions for that problem.

Given $\mathcal{MC}$ and $\mathcal{PS}$, the benchmark dataset $\mathcal{D}$ is created as shown in Algorithm 1. An

Algorithm 1 Benchmark Dataset Generation

```
 1: Initialize dataset $\mathcal{D} \leftarrow \emptyset$
 2: for each student $k \in \{1, 2, \ldots, K\}$ do
 3:     Select misconception $mc^{(k)} \in \mathcal{MC}$
 4:     Initialize problem-code set $PC^{(k)} \leftarrow \emptyset$
 5:     Sample from $\mathcal{PS}$ a set of $N_k$ problem-solution pairs
        $PS^{(k)} = \left\{ \left( p_n^{(k)}, S_n^{(k)} \right) \right\}_1^{N_K}$
 6:     for each $\left( p_n^{(k)}, S_n^{(k)} \right) \in PS^{(k)}$ do
 7:         Sample a correct solution $s_n^{(k)} \in S_n^{(k)}$
 8:         McInject $mc^{(k)}$ in $s_n^{(k)}$ to obtain code $c_n^{(k)}$
 9:         Add problem-code pair $\left( p_n^{(k)}, c_n^{(k)} \right)$ to $PC^{(k)}$
10:     Add $\left( PC^{(k)}, mc^{(k)} \right)$ to dataset $\mathcal{D}$
11: return dataset $\mathcal{D}$ of $K$ misconception-labeled code sets
```

essential component of this procedure is shown at step 8, where a misconception is injected into the correct coding solution of a problem in order to create code that exhibits that misconception. In Section 3.2.1 below we describe the McInject tool that we developed for this purpose.

Table 3.1 summarizes the key statistics of our benchmark dataset. We organized these samples into 339 code sets, or bags of code samples, each containing between $N_k = 4$ to $N_k = 8$ code samples. Importantly, 60 bags contain only correct code samples without any misconceptions, serving as negative examples to test the system's ability to correctly identify when no misconceptions are present.

3.2.1 The McInject Tool

The misconception injection tool McInject takes as input the description of a computational problem p, a correct solution code s, and the description of a programming misconception mc. As output, it produces a code c as a version of s that is modified to exhibit the misconception mc. The code c should look as if written by a student holding misconception mc who tries to solve problem p. The tool is implemented using Claude Sonnet-4.5 with extended thinking, with a structured prompt and in-context learning examples. When a misconception cannot be meaningfully applied to a given solution code, MCINJECT is instructed to indicate so, rather than force inappropriate modifications. An LLM-as-judge evaluation shows that 90.3% of the code samples successfully exhibit their target misconceptions. Note that this is a conservative estimate of MCINJECT's performance, since the LLM-as-judge sometimes discards code samples that exhibit benign misconceptions, where the code is both correct and natural. Further manual evaluation of 88 code samples shows a 96.6% agreement between LLM judgment and human assessment, confirming the reliability of the LLM-as-judge. Appendix B.1 provides detailed descriptions of the tool development, including the prompt, LLM hyper-parameters, and evaluation setting.

3.3 The McMiner Tools

As defined in Section 3.2, in the misconception mining task the input consists of a bag of one or more problem-code pairs from a student, and the output is a potential misconceptions exhibited in the code samples. To solve this task, we developed two variants of an LLM-based mining tool: a *single instance* variant and a *multiple instance* variant.

The single instance variant MCMINER-S proceeds in two stages, as shown in Algorithm 2. First, an LLM is instructed to identify potential misconceptions in each problem-code pair (steps 2 to 4), using the prompt shown in Appendix B.2. Then, out of all the identified misconceptions, the one found in the most problem-code pairs is returned (steps 5 to 10).

In the multiple instance variant MCMINER-M, an LLM is given the entire bag of program-code pairs and is instructed to identify the misconception that is shared by the largest number

Algorithm 2 McMiner-S

Input: Bag of pairs $PC = \{(p_1, c_1), (p_2, c_2), \ldots, (p_N, c_N)\}$ **Output**: Potential misconception $\hat{mc}$ for the bag

```
Initialize set of misconceptions $M \leftarrow \emptyset$
for each problem-code pair $(p_j, c_j) \in PC$ do
    Identify potential misconception $mc_j$ for $(p_j, c_j)$
    If found, add $mc_j$ to $M$
for each misconception $mc \in M$ do
    Let $count(mc) = |\{(p_j, c_j) \in PC | mc_j = mc\}|$
if all counts are 0 then                         ▷ no misconception found
    return $\epsilon$
else                                             ▷ some misconception found
    return $\hat{mc} = \operatorname{argmax}_{mc \in M} count(mc)$
```

Table 3.2: McMiner results for multiple-instance mining (left) and single-instance mining (right). + R indicates that the model had reasoning mode enabled.

	McMiner-M				McMiner-S
Language Model	**Precision**	**Recall**	**F1-Score**	**Accuracy**	**Accuracy**
o3-mini (low-effort)	85.6%	67.5%	75.5%	75.5%	76.9%
o3-mini (medium-effort)	83.3%	70.8%	76.5%	76.9%	**78.5%**
Claude Sonnet-4.5	79.1%	77.7%	78.4%	78.7%	66.4%
Claude Sonnet-4.5 + R	83.8%	77.2%	**80.3%**	**82.0%**	68.7%
Gemini 2.5-flash	79.7%	56.2%	65.9%	61.1%	74.0%
Gemini 2.5-flash + R	77.0%	76.7%	76.8%	76.9%	69.4%

of code samples in the bag, using the prompt shown in Appendix B.2.

Compared to McMiner-S, McMiner-M has the advantage of looking at multiple code samples that may exhibit the same potential misconception pattern, which in theory should enable more precision in identifying misconceptions.

3.4 Experimental Evaluations

We used the benchmark dataset to evaluate the effectiveness of the two McMiner tools when instantiated using three LLMs: OpenAI's o3-mini models, Anthropic's Claude-Sonnet-4.5 models, and Gemini 2.5-Flash models. The experimental setup for each model is detailed in Appendix B.4.

Table 3.2 present the performance of different LLMs in the multiple and single instance mining settings. For computing accuracy, precision, recall, F1-score, we define: *true positives* are cases where the Ground Truth (GT) misconception matches the prediction, or the prediction is a validated novel misconception; *true negatives* are cases where both GT and prediction contain no misconception; *false positives* are cases where the prediction does not match GT and fails validation; and *false negatives* are cases where GT contains a misconception but the model predicted none. Note that when a validated novel misconception is found but GT contains a different misconception, this counts as both a true positive (for the valid discovery) and as a false negative (for missing GT).

Multi-instance mining (MCMINER-M) substantially outperforms single-instance mining (MCMINER-S), with the best model (Claude Sonnet-4.5 + Reasoning) achieving 82.0% accuracy for MCMINER-M compared to 78.5% for the best MCMINER-S model (o3-mini medium-effort). This demonstrates the value of analyzing multiple code samples simultaneously to identify consistent patterns with higher confidence. Enabling reasoning capabilities consistently improves MCMINER-M performance across model families: Claude (78.7% to 82.0%), Gemini (61.1% to 76.9%), and o3-mini (75.5% to 76.9% with medium effort).

The best MCMINER-M model achieves 93.3% accuracy on bags with misconceptions but only 59.3% on correct-only bags, indicating higher false positive rates on correct code.

MCMINER identifies novel misconceptions effectively: Claude Sonnet-4.5 with reasoning discovered 41 novel true positives (12.1% of bags), including: "The student believes that the division operator `/` and integer division operator `//` are interchangeable or that `/` will automatically return an integer when the result is a whole number."

Misconception difficulty varies substantially: syntax errors (e.g., = vs. ==) achieve >95% accuracy, while subtle benign misconceptions (e.g., unnecessary parentheses) and subtle out-of-distribution cases (48% for vowel-based naming) prove more challenging, specially for single-instance mining. Misconceptions that require the LLM to reason from the student's perspective (e.g., operator precedence) achieve 42-50% accuracy. Overall, compared with the single-instance version, multi-instance mining obtains improved performance on more difficult cases.

3.5 Qualitative Analysis

To better understand the performance patterns observed in the experimental evaluations, we conducted qualitative analyses of both the McInject and McMiner tools. These analyses reveal systematic patterns in success and failure cases, providing insights into the capabilities and limitations of LLM-based misconception injection and mining.

We analyzed McInject's generated code samples to understand performance variation across misconception types.

► **McInject Success Patterns.** Several misconceptions achieved 100% exhibit rates. Analysis reveals these fall into two primary categories:

Syntax-based misconceptions produce clear, unambiguous code patterns that directly violate Python syntax rules. For example, (*Student believes that the = operator is used for equality comparison in conditional statements*) and (*Student believes that colons (:) are used for variable assignment*) create syntax errors that are straightforward to inject and verify. Similarly, (*Student believes that a function can be defined without the def keyword*) removes a required keyword, making the misconception immediately apparent.

Clear misuse misconceptions involve well-defined incorrect usage of language constructs where the distinction between correct and incorrect usage is unambiguous. (*Student believes that a print statement must be used to return a value from a function*) exemplifies this pattern – replacing `return` with `print` creates a distinct behavioral signature that is both easy to inject and verify. (*Student believes that functions are called using square brackets like list indexing*) – this similarly produces code with an unmistakable incorrect pattern.

► **McInject Failure Patterns.** Conversely, misconceptions with exhibit rates below 70% reveal two distinct failure modes that highlight the challenges in generating realistic student-like code.

Some misconceptions can manifest in code that is both correct and natural, making it difficult to determine whether the code reflects the misconception or simply represents a valid solution approach. Consider the misconception (*Student believes that loop iteration requires manual counter tracking with an explicit variable to access element indices*) for the problem *"Write a function to find the next smallest palindrome greater than a given number"* MCINJECT generated:

```
import sys
def next_smallest_palindrome(num):
    numstr = str(num)
    i = num + 1
    while i < sys.maxsize:
        if str(i) == str(i)[::-1]:
            return i
```

```
        i += 1
```

This code correctly solves the problem and does exhibit manual counter tracking. However, it is difficult to determine whether a student wrote this code due to holding the misconception or simply because this represents a valid and natural solution approach using a while loop. The LLM-as-a-judge in MCINJECT's validation pipeline tends to be conservative in such cases, labeling these samples as not clearly exhibiting the misconception. While this reduces the exhibit rate, it improves dataset quality by ensuring that retained samples exhibit the misconception more clearly.

This ambiguity pattern particularly affects operator precedence misconceptions, which achieve approximately 50% exhibit rates. These misconceptions require subtle modifications to arithmetic expressions that can produce correct results or produce unnatural code such as writing `num + 2 * 10` instead of `num + 20`. In such cases, MCINJECT tends to focus on emulating the syntactic code structure dictated by the misconception rather than writing code that the student with the misconception would write and genuinely believes would solve the problem correctly.

A second failure mode occurs when MCINJECT produces valid code matching the misconception pattern but lacking semantic plausibility. Successfully injecting subtle misconceptions requires theory of mind, the LLM must authentically adopt the perspective of a student holding that false belief. When this fails, the tool produces code that exhibits the pattern but makes no logical sense.

For instance, for Misconception 18 (*Student believes that in the expression x == a or b, the comparison operator distributes to both operands of the or operator*), MCINJECT generated for the same palindrome problem:

```
import sys
def next_smallest_palindrome(num):
    numstr = str(num)
    for i in range(num+1, sys.maxsize):
        if str(i) == str(i)[::-1] or str(i)[0]:
            return i
```

This code is unrealistic. It makes no sense for a student to check whether `str(i) == str(i)[0]` (checking if a string equals its first character) when they could simply check `len(i) == 1`. Furthermore, the or condition is not needed at all. The program is correct without it. The code syntactically matches the misconception pattern but fails the semantic plausibility test.

Similarly, for *Student believes that chained function calls are evaluated from right to left*, McInject produced:

```
import sys
def next_smallest_palindrome(num):
   numstr = str(num)
   for i in range(num+1, sys.maxsize):
      if str(i).replace(’0’, ’’)[::-1] == str(i).replace(’0’, ’’):
         return i
```

While this attempts to demonstrate order-dependent operations through chained method calls, the code is buggy and nonsensical. Removing zeros from a number before or after reversing is not a correct solution to the palindrome problem and does not reflect a plausible student reasoning path.

▶ **McMiner Patterns.** We now examine the performance patterns and error modes of MCMINER on the benchmark dataset.

A critical evaluation metric for misconception mining tools is their ability to correctly identify when code contains no misconceptions, avoiding false positives that could mislead educators. Analysis of MCMINER-M performance reveals a gap between these two scenarios across different model configurations, as shown in Table 3.3.

Table 3.3: Performance comparison of MCMINER-M across different models on correct-only bags versus bags containing misconceptions. All metrics are based on the 339 total bags (279 bags with misconceptions and 60 correct-only bags).

Model	**Reasoning**	**Correct-Only Accuracy**	**Misc. Bags Accuracy**
CLAUDE SONNET-4.5	✓	59.30%	89.72%
CLAUDE SONNET-4.5	×	41.86%	91.30%
O3-MINI (medium-effort)	–	66.28%	80.63%
O3-MINI (low-effort)	–	75.58%	75.49%
GEMINI 2.5-FLASH	✓	36.05%	90.91%
GEMINI 2.5-FLASH	×	61.63%	60.87%

The results reveal substantial variation in false positive rates across models. Claude Sonnet 4.5 with reasoning achieves 89.72% accuracy on misconception bags but only 59.30% on correct-only bags. Without reasoning, this gap widens (91.30% vs. 41.86%), indicating that reasoning capabilities help reduce false positives. OpenAI o3-mini (low) demonstrates the best balance with nearly equal performance on both bag types (75.58% vs. 75.49%).

Analysis of false positives on correct-only bags reveals systematic patterns in the types of "misconceptions" models incorrectly identify **Inefficiencies as Misconceptions**. This is where models sometimes identify "inefficiencies" as misconceptions, such as flagging manual iteration instead of using built-in functions like `enumerate()` or list comprehensions. While these may represent less idiomatic code, they do not constitute misconceptions about language semantics.

A key strength of McMiner is its ability to discover misconceptions not present in the predefined misconception bank. Using LLM-as-a-judge validation (Section B.5), we identified novel true positives, predicted misconceptions that, while not matching ground truth, accurately describe genuine programming misunderstandings exhibited in the code.

Claude Sonnet 4.5 with reasoning discovered 41 novel true positives across the 339 bags (12.1%), while OpenAI o3-mini (medium effort) found 35 (10.3%).

Example: Division Operator Type Semantics

For a bag containing code implementing tetrahedral number (which is always an integer) calculation, McMiner identified:

> **Predicted Misconception:** "The student believes that the division operator `/` and integer division operator `//` are interchangeable or that `/` will automatically return an integer when the result is a whole number."
>
> **Code Context:**
>
> ```
> def tetrahedral_number(n):
>     return (n * (n + 1) * (n + 2)) / 6
> ```
>
> **Explanation:** In Python 3, the `/` operator always returns a float, even when dividing two integers that result in a whole number. For calculating tetrahedral numbers, which are always integers, using `//` would be more appropriate to return an integer type.

This misconception was not in the original bank but represents a valid and common misunderstanding about Python 3's division semantics.

► **McMiner Error Analysis.** Certain misconceptions in the benchmark proved particularly challenging for McMiner, with several achieving near-zero detection rates. Analysis reveals these share a common characteristic: they are mostly *benign misconceptions*, false beliefs that do not cause bugs but instead reflect stylistic preferences or suboptimal practices.

Low-Performing Misconceptions:

- "Student believes that the `return` statement requires parentheses around its argument" code like `return(x + y)` is syntactically valid and functions correctly
- "Student believes that function parameters automatically change in recursive calls without explicit modification.". This is a harmful misconception that can cause infinite recursion.
- "Student believes that loop iteration requires manual counter tracking with an explicit variable to access element indices.". This benign misconception can often result in functionally correct code that is also natural and lacks unusual features.

These benign misconceptions fail detection because:

1. **Code is functionally correct**: Models trained on code understanding tasks prioritize functional correctness, making them less sensitive to stylistic variations
2. **Patterns lack distinctive signatures**: Unlike misconceptions that produce syntax errors or logical bugs, benign misconceptions produce code indistinguishable from intentional stylistic choices
3. **Ambiguous intent**: Without access to the student's reasoning, it is impossible to defini-

tively determine whether `return(x)` reflects a misconception or a stylistic preference influenced by other programming languages

3.6 McMiner Interface

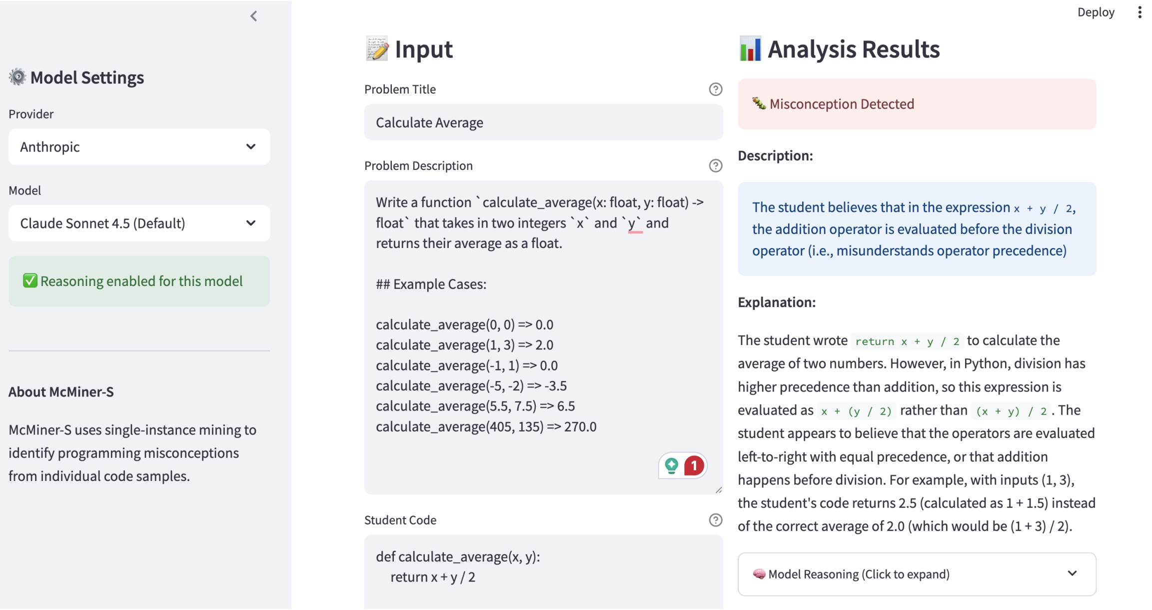
To facilitate the use of McMiner for educators and researchers, we developed an interactive web application using the Streamlit[2] Python library that implements the McMiner-S single-instance mining approach. The interface allows users to input a problem description and student code, then select from multiple state-of-the-art LLMs (Claude, GPT, Gemini) to analyze the code for potential programming misconceptions. The tool, as shown in Figure 3.2, uses the same prompt template and model configurations as our benchmark experiments, automatically enabling reasoning capabilities for compatible models. Results are displayed with structured misconception descriptions, explanations of how they manifest in the code, and expandable reasoning traces from the model. The application loads user API credentials securely from environment variables and provides real-time analysis of student code within seconds.

3.7 Related Work

Traditional automated systems [153, 154, 149] rely on hand-crafted rules targeting a limited set of specific misconceptions. Similarly, modern LLM-based approaches such as [155] focus on classification of logical errors into a predefined set of categories. Overall, targeting only predefined or common misconceptions is a significant limitation, considering that educator beliefs about common errors can diverge significantly from actual student patterns [75]. In parallel, other approaches aim to identify code segments that contain

---

[2] `https://streamlit.io`



Model Settings

Provider
Anthropic

Model
Claude Sonnet 4.5 (Default)

✅ Reasoning enabled for this model

About McMiner-S

McMiner-S uses single-instance mining to identify programming misconceptions from individual code samples.

Input

Problem Title
Calculate Average

Problem Description

Write a function `calculate_average(x: float, y: float) -> float` that takes in two integers `x` and `y` and returns their average as a float.

Example Cases:

calculate_average(0, 0) => 0.0
calculate_average(1, 3) => 2.0
calculate_average(-1, 1) => 0.0
calculate_average(-5, -2) => -3.5
calculate_average(5.5, 7.5) => 6.5
calculate_average(405, 135) => 270.0

Student Code

```
def calculate_average(x, y):
    return x + y / 2
```

Analysis Results

Misconception Detected

Description:

The student believes that in the expression `x + y / 2`, the addition operator is evaluated before the division operator (i.e., misunderstands operator precedence)

Explanation:

The student wrote `return x + y / 2` to calculate the average of two numbers. However, in Python, division has higher precedence than addition, so this expression is evaluated as `x + (y / 2)` rather than `(x + y) / 2`. The student appears to believe that the operators are evaluated left-to-right with equal precedence, or that addition happens before division. For example, with inputs (1, 3), the student's code returns 2.5 (calculated as 1 + 1.5) instead of the correct average of 2.0 (which would be (1 + 3) / 2).

Figure 3.2: Web application interface of the McMiner-S tool. Users are able to input a programming problem and a student implementation, and analyze the code for any potential misconceptions.

logical errors, such as [156, 157], *inter alia*. While knowing the location of logical errors is important, current approaches do not articulate what the actual misconception is about, which can be argued to be equally important from an educational standpoint. In this context, the McMining task is novel in that it requires not only the identification of potentially unknown misconceptions from consistent samples of student code, but also articulating their description in terms of a false belief about a programming concept.

Static analysis tools such as PythonTA [158] help students identify common style and coding errors through automated code inspection. While static analysis is effective at detecting surface-level errors (syntax violations, type mismatches, style inconsistencies), it fundamentally differs from misconception mining in both scope and objective. Static analysis operates on individual code samples to identify detectable errors, whereas misconception mining analyzes patterns across multiple student code submissions (organized into bags) to identify underlying false beliefs about programming language constructs. Critically, not all misconceptions manifest as detectable errors, as a student may hold a false belief that happens to produce correct output on available test cases, and conversely, not all errors indicate misconceptions, as students may make simple typos or violations without harboring false beliefs. Our work focuses on mining the conceptual misunderstandings that cause errors, rather than detecting the errors themselves, enabling targeted pedagogical interventions that address the root cause rather than symptoms.

3.8 Conclusion

This chapter introduces MCMINER, an LLM-based tool for automatically discovering programming misconceptions in student code. Our evaluation on a benchmark of 1,063 code samples exhibiting 67 misconceptions demonstrates strong performance, with the best multi-instance model achieving 82.0% accuracy. Critically, MCMINER can identify emerging and rare student misconceptions not captured in existing taxonomies, discovering 41 novel misconceptions in our evaluation. This capability opens the door to uncovering rare, student-specific misconceptions and enables more personalized learning experiences.

While our benchmark assumes single misconceptions per code sample, real students often exhibit multiple interacting misconceptions. The evaluation is limited to Python, and the LLM-based approach incurs computational costs that may limit large-scale deployment. Classroom deployment also raises pedagogical considerations: over-reliance on automated detection may reduce instructor diagnostic skills or lead to premature intervention before students have the opportunity for self-correction through debugging. Additionally, processing student code through LLM APIs requires consideration of regulations such as FERPA.

CHAPTER 4: REASONING TRAJECTORIES FOR SOCRATIC DEBUGGING

In Socratic debugging, instructors guide students towards identifying and fixing a bug on their own, instead of providing the bug fix directly. Most novice programmer bugs are caused by programming misconceptions, namely false beliefs about a programming concept. In this context, Socratic debugging can be formulated as a guided Reasoning Trajectory (RT) leading to a statement about the program behavior that contradicts the bug-causing misconception. Upon reaching this statement, the ensuing cognitive dissonance leads the student to first identify and then update their false belief. This chapter, introduces the task of reasoning trajectory generation, together with a dataset of debugging problems manually annotated with RTs. We then describe LLM-based solutions for generating RTs and Socratic conversations that are anchored on them. A large-scale LLM-as-judge evaluation shows that frontier models can generate up to 91% correct reasoning trajectories and 98.7% valid conversation turns.[1]

Chapter Contributions. ★

1. Introduce the task of reasoning trajectory generation, a novel approach to Socratic debugging where instructors guide students through structured inference steps that culminate in statements contradicting bug-causing misconceptions.

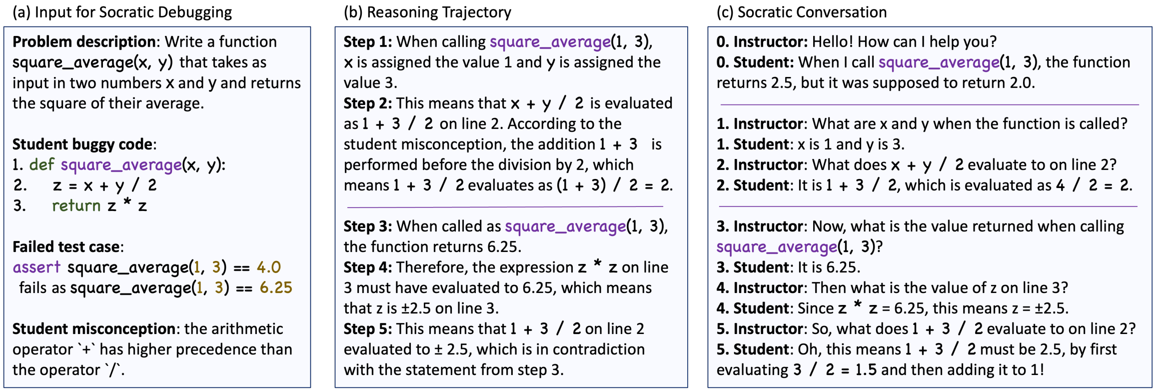
---

[1] This chapter is based on a preprint publication that is being prepared for submission to a peer-reviewed conference. A preprint version of the manuscript is available here [88].

2. Develop a two-stage pipeline that separates planning (RT generation) from articulation (SC generation), enabling systematic generation of Socratic guidance where each teacher utterance elicits a specific reasoning step from the student.
3. Creates a benchmark dataset of 227 problem-solution-misconception triplets spanning 40 unique misconceptions, with LLM-generated reasoning trajectories containing over 22K reasoning steps across 14 model configurations.
4. Design and validate an LLM-as-judge evaluation methodology that achieves 76.6% agreement with human judgment on RT validity and 96.5% agreement on Socratic turn quality, enabling scalable assessment of generated trajectories and their corresponding Socratic conversations.
5. Demonstrate that frontier LLMs can generate high-quality Socratic debugging guidance, with the best models achieving 91.1% valid reasoning trajectories and 98.7% valid conversations, while revealing that extended reasoning capabilities improve performance for some models but not others.

4.1 Introduction and Motivation

I introduce the SOCRATIC DEBUGGING task in Chapter 2 as a conversation between a Student and an Instructor aimed at helping the student fix buggy code through guided inquiry. During Socratic dialogue an instructor's questions may probe a student's existing knowledge or assumptions; guide attention to relevant aspects of a complex problem; or encourage discovery of general principles through the consideration of alternative solutions



(a) Input for Socratic Debugging

Problem description: Write a function `square_average(x, y)` that takes as input in two numbers x and y and returns the square of their average.

Student buggy code:

```
1. def square_average(x, y):
2.     z = x + y / 2
3.     return z * z
```

Failed test case:

`assert square_average(1, 3) == 4.0`
fails as `square_average(1, 3) == 6.25`

Student misconception: the arithmetic operator `+` has higher precedence than the operator `/`.

(b) Reasoning Trajectory

Step 1: When calling `square_average(1, 3)`, x is assigned the value 1 and y is assigned the value 3.
Step 2: This means that `x + y / 2` is evaluated as `1 + 3 / 2` on line 2. According to the student misconception, the addition `1 + 3` is performed before the division by 2, which means `1 + 3 / 2` evaluates as `(1 + 3) / 2 = 2`.

Step 3: When called as `square_average(1, 3)`, the function returns 6.25.
Step 4: Therefore, the expression `z * z` on line 3 must have evaluated to 6.25, which means that z is ±2.5 on line 3.
Step 5: This means that `1 + 3 / 2` on line 2 evaluated to ± 2.5, which is in contradiction with the statement from step 3.

(c) Socratic Conversation

0. Instructor: Hello! How can I help you?
0. Student: When I call `square_average(1, 3)`, the function returns 2.5, but it was supposed to return 2.0.

---

1. Instructor: What are x and y when the function is called?
1. Student: x is 1 and y is 3.
2. Instructor: What does `x + y / 2` evaluate to on line 2?
2. Student: It is `1 + 3 / 2`, which is evaluated as `4 / 2 = 2`.

---

3. Instructor: Now, what is the value returned when calling `square_average(1, 3)`?
3. Student: It is 6.25.
4. Instructor: Then what is the value of z on line 3?
4. Student: Since `z * z` = 6.25, this means z = ±2.5.
5. Instructor: So, what does `1 + 3 / 2` evaluate to on line 2?
5. Student: Oh, this means `1 + 3 / 2` must be 2.5, by first evaluating `3 / 2 = 1.5` and then adding it to 1!

Figure 4.1: Socratic debugging example: (a) the input specifies the problem, the buggy code, the failed test case, and the student misconception that caused the bug; (b) a reasoning trajectory ending with a statement that contradicts the misconception; (c) a Socratic conversation that follows the reasoning trajectory and ends with a belief update.

or counterexamples [159]. Through its emphasis on active inquiry, in-context reasoning about evidence, and repeated retrieval of relevant concepts from memory, Socratic dialogue engages students in deep thinking and meaningful integration of new knowledge, which can greatly improve their acquisition of generalizable skills and ultimately their learning outcomes [160].

This chapter introduces tools that support instructors to first *plan* and then *articulate* Socratic conversations for debugging. This two-phase approach is inspired by dialogue systems literature, where decoupling natural language generation into separate planning and realization phases has been shown to outperform end-to-end architectures [161]. Our key insight is that effective Socratic debugging conversations should guide students along a *reasoning trajectory*, a carefully designed sequence of inferences about code behavior that leads to a statement directly contradicting the student's misconception.

In a Socratic debugging approach, it is essential that the student himself realizes which of his programming beliefs are false, i.e., misconceptions. By guiding the student to discover and fix a misconception on his own, the instructor also maximizes the likelihood that the fixed belief will endure over time and not revert to the initial false belief. To achieve this aim, we propose that the Instructor guide the student along a sequence of inferences about the code behavior for a failed test case. The *reasoning trajectory* is designed such that the last inference step proves a statement that is in direct contradiction with the student's misconception. This overt contradiction between the false belief and the actual code behavior is expected to create a strong *cognitive dissonance* [162] for the student, who consequently not only realizes which of his beliefs is false, but also corrects it on his own, as shown in the example in Figure 4.1. In general, the psychological discomfort associated with cognitive dissonance has been found empirically to be extremely motivating in terms of triggering learning processes that seek to resolve the dissonance [163, 164]. As described in [165], placing learners in a state of cognitive dissonance is ideal for learning in problem-solving scenarios, where the intrinsic human need for consistency and equilibrium leads to a constant process of examining new information and updating existing knowledge structures [166].

4.2 Task Definition

As shown in Figure 4.1(a), the input to the Socratic debugging task consists of a problem description, the student's buggy code, a failed test case, and the student's misconception that caused the bug. Consistent with the aforementioned aim of guiding the student towards discovering his own misconception, we approach the task of Socratic debugging as a pipeline of two main subtasks:

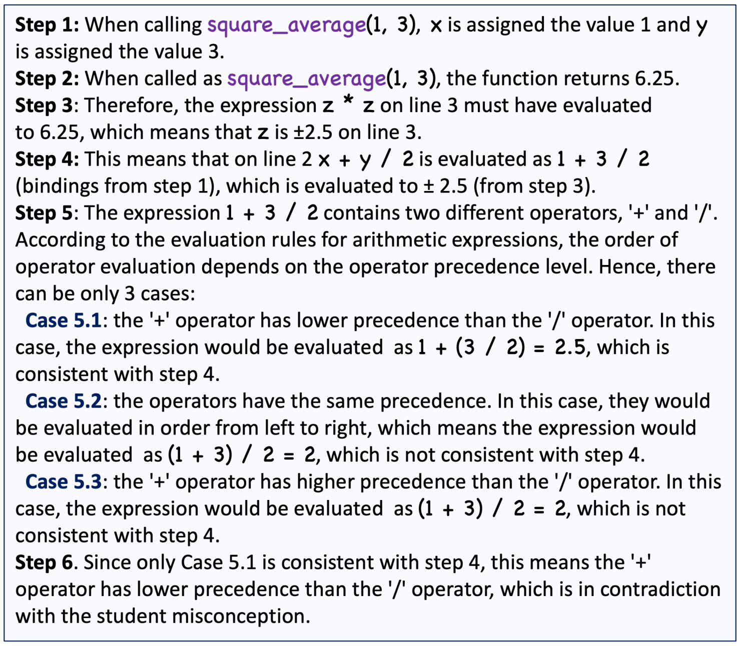
1. **Reasoning Trajectory (RT)**: In the first step, a reasoning trajectory is generated as a sequence of inference steps such that the statement proven in the last step contradicts

the misconception or provides a counterexample to the misconception, as shown in Figure 4.1(b).

2. **Socratic Conversation (SC)**: In the second step, a Socratic conversation is generated step by step, such that each RT step is associated with an Instructor turn followed by a Student turn, where the instructor's question aims to elicit from the student the statement proven at that step, as illustrated in Figure 4.1(c).

The reasoning trajectory shown in Figure 4.1(b) is structured in two parts. In the first part, the reasoning steps lead to showing a statement of the student's misconception for the failed test case, namely that the expressions 1 + 3 / 2 evaluates as 2. In the second part, the reasoning proceeds backwards from the returned value in order to infer a statements that contradicts the misconception statement, namely that 1 + 3 / 2 evaluates to ± 2.5. Note that this is not the only way of articulating an RT that ends with a statement contradicting the misconception. Figure 4.2 shows an alternative RT where the reasoning steps end with a statement that is the opposite of the misconception statement. Thus, while the RT from Figure 4.1 (b) can be seen as providing a counterexample to the misconception statement by instantiating it for a particular failed test case, the RT in Figure 4.2 does not instantiate the misconception statement and instead proves a statement that contradicts the misconception statement in the general case. Given that the first type of RTs are generally shorter, in this chapter we focus on generating RTs that derive counterexamples to the student misconception. The LLM-based approach for generating reasoning trajectories is described in Section 4.4.1.

Once a reasoning trajectory is generated, it is used step by step to generate a corresponding Socratic conversation. As shown in Figure 4.1(c), the Socratic conversation is structured in



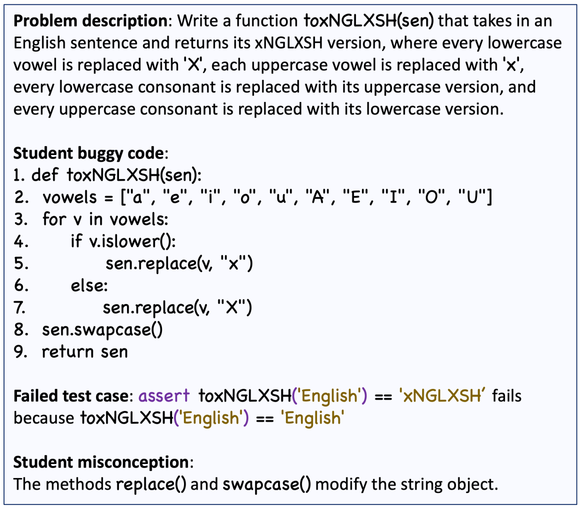
Step 1: When calling `square_average(1, 3)`, `x` is assigned the value 1 and `y` is assigned the value 3.
Step 2: When called as `square_average(1, 3)`, the function returns 6.25.
Step 3: Therefore, the expression `z * z` on line 3 must have evaluated to 6.25, which means that `z` is ±2.5 on line 3.
Step 4: This means that on line 2 `x + y / 2` is evaluated as `1 + 3 / 2` (bindings from step 1), which is evaluated to ± 2.5 (from step 3).
Step 5: The expression `1 + 3 / 2` contains two different operators, '+' and '/'. According to the evaluation rules for arithmetic expressions, the order of operator evaluation depends on the operator precedence level. Hence, there can be only 3 cases:
Case 5.1: the '+' operator has lower precedence than the '/' operator. In this case, the expression would be evaluated as `1 + (3 / 2) = 2.5`, which is consistent with step 4.
Case 5.2: the operators have the same precedence. In this case, they would be evaluated in order from left to right, which means the expression would be evaluated as `(1 + 3) / 2 = 2`, which is not consistent with step 4.
Case 5.3: the '+' operator has higher precedence than the '/' operator. In this case, the expression would be evaluated as `(1 + 3) / 2 = 2`, which is not consistent with step 4.
Step 6. Since only Case 5.1 is consistent with step 4, this means the '+' operator has lower precedence than the '/' operator, which is in contradiction with the student misconception.

Figure 4.2: Alternative reasoning trajectory for the input from Figure 4.1(a).

three parts. The first part contains a generic, initial statement from the instructor, while the student's turn describes the failed test case. The turns in the second and third parts map to the steps in the first and second parts of the RT, respectively. For each RT step, the instructor asks a question that aims to guide the student towards articulating the statement proven at that step. Note that although we generate a Socratic turn for each RT step, it is possible for the instructor to skip one or more steps if she determines that the student is capable of making one or more inferences on his own, without guidance. For example, right after turn 3, the instructor can choose to skip turn 4 and go directly to turn 5. The LLM-based approach for generating reasoning trajectories is described in Section 4.4.2.

4.2.1 Simplification

As shown in Figure 4.1(b), the reasoning trajectory is structured in two parts: the first part leads to an instance of the misconception, whereas the second part leads to a statement that



Problem description: Write a function toxNGLXSH(sen) that takes in an English sentence and returns its xNGLXSH version, where every lowercase vowel is replaced with 'X', each uppercase vowel is replaced with 'x', every lowercase consonant is replaced with its uppercase version, and every uppercase consonant is replaced with its lowercase version.

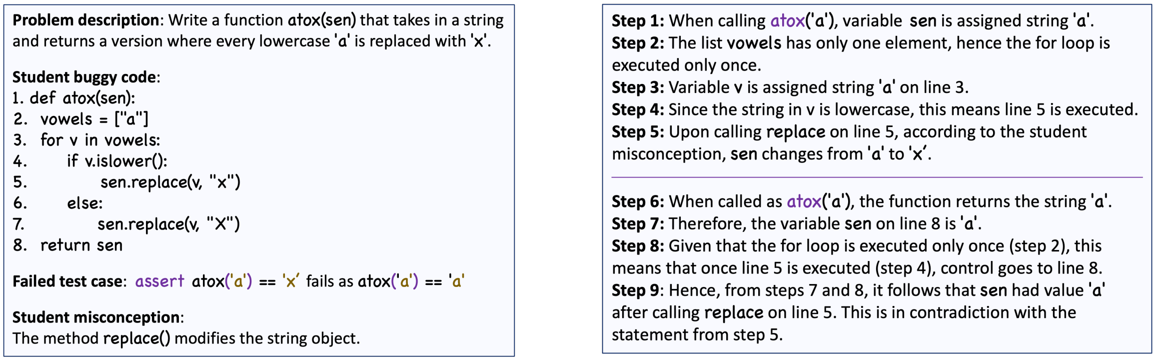
Student buggy code:

```
1. def toxNGLXSH(sen):
2.   vowels = ["a", "e", "i", "o", "u", "A", "E", "I", "O", "U"]
3.   for v in vowels:
4.       if v.islower():
5.           sen.replace(v, "x")
6.       else:
7.           sen.replace(v, "X")
8.   sen.swapcase()
9.   return sen
```

Failed test case: `assert toxNGLXSH('English') == 'xNGLXSH'` fails because `toxNGLXSH('English') == 'English'`

Student misconception:
The methods **replace()** and **swapcase()** modify the string object.

(a) The original input specification.



Problem description: Write a function **atox(sen)** that takes in a string and returns a version where every lowercase '**a**' is replaced with '**x**'.

Student buggy code:

```
1. def atox(sen):
2.   vowels = ["a"]
3.   for v in vowels:
4.       if v.islower():
5.           sen.replace(v, "x")
6.       else:
7.           sen.replace(v, "X")
8.   return sen
```

Failed test case: `assert atox('a') == 'x'` fails as `atox('a') == 'a'`

Student misconception:
The method **replace()** modifies the string object.

Step 1: When calling atox('a'), variable **sen** is assigned string '**a**'.
Step 2: The list **vowels** has only one element, hence the for loop is executed only once.
Step 3: Variable v is assigned string '**a**' on line 3.
Step 4: Since the string in v is lowercase, this means line 5 is executed.
Step 5: Upon calling **replace** on line 5, according to the student misconception, **sen** changes from '**a**' to '**x**'.

Step 6: When called as atox('a'), the function returns the string '**a**'.
Step 7: Therefore, the variable **sen** on line 8 is '**a**'.
Step 8: Given that the for loop is executed only once (step 2), this means that once line 5 is executed (step 4), control goes to line 8.
Step 9: Hence, from steps 7 and 8, it follows that **sen** had value '**a**' after calling **replace** on line 5. This is in contradiction with the statement from step 5.

(b) The simplified input for the original in (a). (c) The RT for the simplified input in (b).

Figure 4.3: Simplification process: (a) original input with multiple misconceptions and complex execution, (b) simplified input focusing on a single misconception, and (c) the corresponding reasoning trajectory for the simplified input.

contradicts it. It is important for both parts in this reasoning process to be short, otherwise a long and complicated reasoning trajectory can place a significant cognitive burden on the student, which will defeat the aim of Socratic guidance. Therefore, to keep the complexity of the reasoning traces at a feasible level, we envision a simplification process where the original problem description, code, and failed test case are simplified such that (a) they focus on the code behavior that instantiates the misconception, while (b) they stay as close to the original as possible. In Figure 4.3a we show an example input, where formulating a reasoning trajectory would be overly complicated due to the many calls to the function about which the student has a misconception, and the length of the input string. Furthermore, the student has two misconceptions, whereas by definition a reasoning trace corresponds to just one misconception. While it is possible to merge the two misconceptions into a general misconception that subsumes both, e.g., "string methods can modify the string object", it is easier for the student to address concrete misconceptions, one at a time. Correspondingly, the original task is simplified as shown in Figure 4.3b, whereas the corresponding RT is shown in in Figure 4.3c.

4.3 Dataset

We leverage the problem-solution and misconception dataset introduced in the MCMINING dataset from [87] containing 501 problems, 558 solutions, and a subset of 40 bug-inducing misconceptions. Given the 558 coding solutions and an input misconception, we developed a construct-based pairing algorithm that identifies the most suitable solutions that rely on the programming concept referenced by the misconception, ending with 250 solutions connected with misconceptions through a programming construct. The pseudocode is detailed in Appendix C.2.

The $\langle$*problem*, *solution*, *misconception*$\rangle$ triplets are then used as input to the MCINJECT tool introduced in [87], which generates buggy code samples by injecting the misconception in the correct solution. To ensure the misconception is correctly exhibited in the buggy code, we use MCINJECT with up to 3 refinement iterations. The refinement process uses an LLM-as-judge to determine whether the buggy code exhibits a misconception or not, providing feedback to MCINJECT if the code does not yet exhibit the misconception [87].

Table 4.1: Overall dataset statistics. Each of the 14 LLM configurations generated reasoning trajectories for all 227 problems, with the number of steps for each model shown in Table 4.2.

Component	**Count**
Problems	501
Solutions	558
Misconceptions	40
⟨Problem, Solution, Misconception⟩ triplets	227
Handwritten	
Reasoning Trajectories	10
Total RT steps	57
LLM-Generated	
LLM configurations	14
Total RT steps	22,506

When used in this way, MCINJECT generated 250 corrupted code samples. Of these, 43 samples were filtered out: 17 due to not being buggy (i.e., they passed all the unit tests), and 26 due to not exhibiting the misconception. To the remaining 207 samples we added 20 handwritten samples, yielding a final dataset of 227 buggy code samples exhibiting an associated misconception. For each buggy code sample, we use an LLM connected to a code execution tool to identify and describe the simplest test case that the buggy code fails. This procedure is described in more detail in Appendix C.4.

The problem description, buggy code sample, failed test case description, and misconception description were then used as input for the Socratic debugging pipeline, where first a reasoning trajectory is generated (Section 4.4.1), and then as input for generating a Socratic conversation (Section 4.4.2). We use 14 different LLM configurations to generate reasoning trajectories and Socratic conversations, as described in detail in Appendix C.3. The overall statistics of the dataset are summarized in Table 4.1.

4.4 Socratic Debugging Pipeline

The generation of Socratic debugging conversations is implemented as a pipeline of two steps. First, a reasoning trajectory is generated that starts from the failed test case and ends with a correct statement about the buggy code behavior that contradicts the student's misconception. Then, the RT is used as a plan for generating a Socratic conversation, where each reasoning step is associated a Socratic turn composed of an Instructor utterance followed by a Student utterance.

4.4.1 Reasoning Trajectories

Given a problem description, the buggy code, a failed test case, and a misconception, an LLM is instructed to generate a sequence of deductive reasoning steps that culminate in a statement contradicting the student's false belief.

Figure 4.4 shows the prompt template used for RT generation.[2] We employ a 2-shot prompting approach, which includes two worked examples and structured input and output formats. The prompt emphasizes five core principles that guide the generation process: (1) strict deductive reasoning with no logical leaps or abductive inferences; (2) consistency with the student's misconception, avoiding the use of programming knowledge that would contradict their false belief, e.g., if the student believes that `range(n)` starts at 1, the RT should not use that fact that `range(n)` starts at 0; (3) exclusive focus on contradicting the misconception rather than providing fixes; (4) starting from observable facts in the failed test case; and (5) sequential reasoning steps with explicit citation of premises.

These principles ensure that generated RTs maintain logical rigor and focus on deducing

---

[2] Complete versions can be seen in the GitHub repository.

Your Task

You will be given a problem description, buggy code, a failed test case, and a student misconception. Your task is to write a reasoning trajectory: a sequence of rigorous, deductive reasoning steps that prove a statement contradicting the misconception.

Core Principles

1. **Strictly deductive**: Each step must be a necessary logical consequence of previous steps, correct programming language knowledge, and observable facts.
2. **Consistent with misconception**: Do not assume programming knowledge that contradicts the student's false belief.
3. **Focus on disproving misconception**: End when reaching a statement that contradicts the misconception. Do not show the correct fix.
4. **Start from failed test**: Begin with observable facts from the failed test case and trace program state throughout execution.
5. **Sequential labeling**: Label steps as A.1, A.2, ..., A.n. Reference non-adjacent prior steps when used.

Input Format

```
<problem>[problem_description]</problem>
<bug_code>[buggy_code]</bug_code>
<failed_test>[failed_test]</failed_test>
<misconception>[misconception]</misconception>
```

Output Format

```
Step A.1: [Observable fact(s) from failed test]
...
Step A.k: [Deduced fact(s) using previous steps]
...
Step A.n: [Statement contradicting misconception]
```

Figure 4.4: Reasoning trajectories prompt template. The full template includes worked examples demonstrating code tracing and proof techniques such as loop invariants.

a statement that contradicts the misconception. Each inference step must follow necessarily from previously established facts and correct knowledge of Python programming that does not contradict the student's misconception. By requiring consistency with the misconception at every intermediate step, we ensure the reasoning steps can be achieved by students who hold that false belief, making the eventual contradiction at the last step more impactful in terms of the cognitive dissonance that it produces.

4.4.2 Socratic Turns

Your Task

You will be given a Reasoning Trajectory (RT), which is a sequence of reasoning steps ending with a statement that disproves a student's misconception. Your task is to write a Socratic conversation between a Teacher and a Student that guides the student to articulate, at each turn, the statement proven at that RT step. The Teacher should not provide statements directly but ask questions that prompt the student to infer them independently.

Guidelines

- **Natural conversation**: Teacher utterances should be direct, clear, and concise. Avoid phrases like "That's an interesting point" or "Good question."
- **Socratic approach**: Ask open-ended questions that require reasoning. Do not state the inference and ask for confirmation.
- **RT correspondence**: Each Teacher utterance prompts step A.X, and each Student response corresponds to A.X.

Formatting and Structure

- Use `Teacher:` and `Student:` as speaker labels
- Conversation begins with Teacher inquiring about the issue

Input Format

```
<problem>[problem_description]</problem>
<buggy_code>[buggy_code]</buggy_code>
<failed_test>[failed_test]</failed_test>
<misconception>[misconception]</misconception>
<rt>[reasoning_trajectory]</rt>
```

Figure 4.5: Prompt template for Socratic conversation generation. The full template includes a worked example demonstrating the correspondence between RT steps and dialogue turns.

Building on the generated reasoning trajectories, we approach Socratic conversation generation as a sequential dialogue turn generation where each RT step anchors an instructor-student exchange. The instructor's utterances are intended to guide the student to make the inferences described at each RT step, instead of providing the inference step directly to the student. For example, if the RT step proves that `range(1)` must have produced the value 0, the teacher should ask a question like *"Where did the value 0 come from?"* rather than *"Isn't it true then that* `range(1)` *must have produced the value 0?"*.

Figure 4.5 shows the prompt template for Socratic conversation generation. We employ a 1-shot prompting approach that includes a worked example showing the complete conversation associated with a reasoning trajectory. The prompt takes as input the complete reasoning trajectory along with the problem specification. The generated conversations follow a natural dialogue structure where the teacher begins by inquiring about the encountered issue, and subsequent turns systematically work through each RT step. Each teacher utterance corresponds to one RT step, aiming to elicit from the student the statement proven at that step. This one-to-one correspondence with the underlying reasoning trajectory ensures that the dialogue maintains logical coherence while preserving the pedagogical value of instructor-guided discovery.

4.5 Experimental Evaluation

We benchmark six state-of-the-art LLMs on their ability to generate valid reasoning trajectories and Socratic conversations: `GPT-5`, `GPT-5-mini`, `Claude Sonnet-4.5`, `Claude Haiku-4.5`, `Gemini 2.5-flash`, and `Gemini 2.5-pro`. The 6 LLMs are evaluated in 14 total configurations with varying levels of reasoning and different hyperparameters, as described in detail in Appendix C.3. All experiments leverage the API from the respective LLM providers.

4.5.1 LLM-as-Judge Methodology

The sheer number of generated RT steps, over 22K as indicated in Table 4.1, prohibits manual evaluation. Consequently, for both RT and Socratic conversation evaluation, we turn to using an LLM-as-judge approach, where:

1. A suitably instructed LLM is first shown to be a reliable evaluator by manually verifying its decision on a small subset of examples (Section 4.5.1.1).
2. The LLM is then deployed to automatically evaluate all generated trajectories and conversations (Section 4.5.1.2).

A priori, using the LLM-as-judge for LLM-based generations is sensible considering that, in general, *verification is much easier than generation*, e.g., determining whether a sequence of reasoning steps is sound is much easier than generating a sequence of reasoning steps that disproves a misconception.

4.5.1.1 Evaluating the LLM-as-Judge

To evaluate the reliability of the LLM-as-judge, we conducted a manual evaluation of the LLM-as-judge output on a subset of 30 RT samples, 10 from each of three models: Claude Sonnet-4.5 with reasoning, Gemini 2.5-pro with reasoning, and GPT-5 with medium reasoning effort. For each sample, we generated reasoning trajectories and Socratic conversations using the model configurations specified in Appendix C.3. We then evaluated these outputs using Claude Sonnet-4.5 with extended thinking as the LLM judge, by applying the evaluation criteria described in Appendices C.5.1 and C.5.2. One of the authors then independently evaluated the same 30 samples for RT validity. Correspondingly, we observed a 76.7% agreement between the LLM judge and the human on reasoning trace evaluation. We also use a subset of 88 teacher utterances to manually evaluate the Socratic turn quality using the same criteria as the LLM-as-judge. Correspondingly, we observed a 96.6% agreement on Socratic turn evaluation.

In RT validation, the LLM judge occasionally struggles with detecting when RTs rely on programming knowledge that contradicts misconceptions and misses technical inaccuracies in terminology (e.g., using "conditional expression" to mean a boolean condition when it actually refers to Python's ternary operator `x if C else y`). In Socratic turn validation, the judge demonstrates strong reliability on clear-cut cases and consistently detects and penalizes teacher utterances consisting of rhetorical questions seeking confirmation. The judge occasionally makes evaluation mistakes whereby it penalizes useful conversational framing, e.g., *"Let's trace through the code"*, by motivating that it is not relevant to eliciting the target reasoning step from the student. For detailed failure patterns in both RT and Socratic turn evaluation, see Appendices C.8.1 and C.8.2, respectively.

4.5.1.2 Using the LLM-as-Judge

To evaluate the RT generation step, we employ the LLM-as-judge approach with structured criteria across four major categories: logical soundness, step construction, precision, and focus, where a correct RT must satisfy all criteria. We then compute the percentage of correct RTs for each model. For more details on the RT evaluation setup, including the evaluation of the LLM-as-judge itself, see Appendix C.5.1.

To evaluate the quality of generated Socratic turns, we employ an LLM-as-judge approach as well, with two key criteria: whether the teacher utterance elicits the correct inference from the corresponding RT step, and whether it avoids stating that inference directly. For a teacher Socratic utterances to be deemed correct, it must satisfy both criteria. We then compute the percentage of valid Socratic turns for each model. Lastly, we compute the percentage of valid Socratic conversations for each model, where a valid conversation must have all valid teacher utterances grounded in the corresponding RT step. For more details on the methodology for Socratic conversation evaluation, see Appendix C.5.2.

4.5.2 Results and Discussion

The results from all 14 LLM configurations are summarized in Table 4.2 and reveal several key findings. First, reasoning trajectory quality varies considerably, with GPT-5 achieving the highest validity rates between $85 - 91\%$, while generating relatively concise trajectories. Notably, extended reasoning capabilities do not uniformly improve performance. Claude models benefit substantially from reasoning mode: Claude Sonnet-4.5 with $+6.6\%$ in RT validity and Claude Haiku-4.5 with $+16.3\%$ in RT validity. Similarly, GPT-5 models perform better with increased reasoning effort. In contrast, the results from Gemini models are mixed, with 2.5-flash performing slightly worse when reasoning is enabled, with -0.8% in RT validity, while generating more reasoning steps.

We also observe a slight inverse relationship between trajectory length and validity: GPT-5 medium-effort produces the fewest total steps (1,271) and achieves the highest RT validity (91.1%), while Claude Haiku-4.5 without reasoning generates the most steps (1,962) but has the lowest validity (62.6%). This is somewhat to be expected, given that the more reasoning steps are contained in an RT, the more chances for one of them to be invalid, which then, according to our evaluation methodology, invalidates the entire RT.

Table 4.2: Performance of language models on reasoning trajectory generation and Socratic conversation generation. RT Steps shows total steps across all 227 samples. Valid RTs measures the percentage of reasoning trajectories that satisfy all correctness criteria. Valid Convs measures whether all teacher turns in a conversation are grounded in the RT. Valid Turns (also referred to as Grounded Turns) measures the percentage of individual teacher turns that are properly grounded in the corresponding RT step.

Language Model	**Reasoning**	**RT Steps**	**% Valid**		
			RTs	**Convs**	**Turns**
GPT-5 (minimal-effort)	✓	1,577	85.0%	94.3%	98.6%
GPT-5 (low-effort)	✓	1,488	90.7%	**98.7%**	**99.4%**
GPT-5 (medium-effort)	✓	1,271	**91.1%**	94.8%	98.5%
GPT-5-mini (minimal-effort)	✓	1,826	68.3%	85.0%	96.9%
GPT-5-mini (low-effort)	✓	1,453	59.5%	92.1%	98.0%
GPT-5-mini (medium-effort)	✓	1,351	68.9%	95.9%	98.7%
Claude Sonnet-4.5	×	1,792	80.6%	89.0%	97.4%
Claude Sonnet-4.5	✓	1,776	87.2%	92.5%	97.9%
Claude Haiku-4.5	×	1,962	62.6%	68.3%	93.2%
Claude Haiku-4.5	✓	1,738	78.9%	81.5%	95.7%
Gemini 2.5-flash	×	1,379	83.5%	86.1%	97.4%
Gemini 2.5-flash	✓	1,826	82.7%	85.8%	96.7%
Gemini 2.5-pro	×	1,439	77.2%	78.3%	94.9%
Gemini 2.5-pro	✓	1,628	85.3%	78.7%	95.5%

Generally, once a reasoning trajectory is generated, the Socratic conversation generation process is relatively straightforward and consistent across different models. Most LLMs are able to generate valid Socratic utterances grounded in the input reasoning trajectory, and they consistently do so throughout an entire conversation.

Qualitative analysis of generated reasoning trajectories reveals that successful RTs exhaustively eliminate alternative possibilities, use concrete execution tracing, and end with clear contradictions. Failure modes include relying on knowledge that contradicts misconceptions and employing abductive rather than reasoning. These patterns are detailed in Appendix C.6.

Qualitative analysis of generated Socratic utterances reveals that LLMs demonstrate accurate RT step alignment, with no observed cases of questions eliciting entirely different reasoning steps than intended. Models successfully integrate facts from prior reasoning steps into coherent questions and employ implicit elicitation, where questions prime students to provide complete logical steps beyond what is explicitly requested. For instance, asking *"What expression is evaluated on line 5?"* implicitly elicits both the abstract expression (e.g. `x = 1 + 5`) and its concrete evaluation (`x = 6`), without requiring separate prompts for each component. A notable failure pattern includes teacher utterances occasionally stating conclusions directly rather than prompting the student to derive them. These patterns are detailed in Appendix C.7.

4.6 Related Work

Scaffolding enables learners to achieve goals through guided efforts [48], and Socratic Questioning (SQ) represents a conversational form of scaffolding where a knowledgeable person helps learners solve problems beyond their current abilities [48, 49, 50]. Wood [47] identified two key questioning types: funneling, which guides learners toward solutions through sequential questions, and focusing, which directs attention to important problem aspects and encourages reflection [47, 113, 114]. While students can complete programming exercises yet struggle to explain their code [115], Tamang et al. [116] demonstrated that Socratic methods effectively improve code comprehension. However, the impact of Socratic questioning on debugging learning outcomes remains unexplored.

Prior AI work in programming education includes intelligent tutoring systems (ITS) and learning support systems that provide automated feedback, generate exercises, and create code explanations [21]. Most ITS models use pre-LLM methods like action-rules and Bayesian networks [117, 118, 119, 120]. Recent work has shown that computer-based scaffolding techniques have a moderate impact on STEM learning [56], with approaches like automatically generating Socratic questions for math problems using fine-tuned language models [26, 27, 28]. Furthermore, several open source LLMs have been fine-tuned on a large amount of synthetic tutoring conversations in mathematics [29] and over 100,000 hours of real tutoring conversations in multiple subjects [30]. Planning in dialogue systems literature has shown that decoupling generation into separate planning and realization phases performs better than end-to-end approaches [161].

Automated hint generation systems aim to assist programming students through instant feedback using techniques like extracting common bugs [124], analyzing peer data patterns [125, 126], and generating custom solution paths [127, 76]. AI tutoring for formal proving in mathematics such as the LeanTutor [167], rely on generating three types of hints: an identification of the error, a single guiding question, an explicit suggestion for the next step, and does not engage in a complete Socratic conversation. Lu and Krishnamurthi [168] present an approach to identifying and correcting student misconceptions about programming language behavior through "misinterpreters", pre-programmed interpreters that can deterministically detect misconceptions about programming language semantics. Their SMoL Tutor uses refutation texts to explicitly address these misconceptions during MCQ quizzes.

Our approach focuses on the diagnosis and correction of misconceptions in buggy code through complete Socratic dialogue. Unlike prior work, we plan Socratic conversations such that they engage the student in a particular type of reasoning about the buggy code behavior, where they are guided towards inferring a correct statement about the actual code execution that conflicts with their misconception. As argued in Section 4.1, reaching this moment of cognitive dissonance is important in that it is expected to trigger a Eureka moment for the student, where they suddenly realize which of their programming beliefs is false, followed by an enduring belief update that fixes the misconception.

4.7 Conclusion

This chapter introduces a novel formulation of Socratic debugging, where the teacher utterances aim to follow a reasoning trajectory that starts from a failed test case, and upon a sequence of inference steps reaches a correct statement about the program that is in contradiction with the student misconception that caused the bug. Upon reaching this statement, the student is expected to experience a strong cognitive dissonance, which then entails an enduring belief update. To support development and evaluation, we created a dataset of 227 problems paired with buggy solutions and the corresponding bug-causing misconception. A large scale LLM-as-judge evaluation of over 22K reasoning trajectory steps and their associated Socratic utterances shows that frontier models can achieve up to 91% trajectory validity and 98.7% conversation validity. Overall, through carefully orchestrated moments of cognitive dissonance, the proposed automated Socratic guidance approach can be of significant benefit to instructors seeking to help students durably fix their programming misconceptions.

CHAPTER 5: CONCLUSION

This dissertation presents a computational framework for automated Socratic debugging conversations in novice programming environments. The framework makes three key contributions through an integrated pipeline: establishing evaluation standards via the Socratic Debugging Benchmark, systematically identifying misconceptions through the McMining framework, and generating well-defined, principled dialogue through Reasoning Trajectories. Together, these contributions establish both the computational framework and conceptual foundations necessary for building automated Socratic agents capable of guiding students to discover and correct programming misconceptions through effective dialogue.

The Socratic Debugging Benchmark dataset discussed in Chapter 2 enables a systematic evaluation of Socratic debugging systems that for the first time accounts for the fact that, at each point in a conversation, there may be multiple valid Socratic utterances. This is important because the typical evaluation standard in dialogue systems relies on single-reference annotations where only one correct response exists per turn, which fundamentally misrepresents the nature of Socratic guidance. The multi-reference annotations enable systematic evaluation using multi-reference precision and recall metrics. Comprehensive evaluation demonstrates that while recent frontier models achieve substantial performance improvements, they still fall short of human performance, indicating significant opportunity for further research. The McMining framework discussed in Chapter 3 shows the capability to discover novel misconceptions beyond predefined taxonomies, providing the essential first step of identifying false beliefs that students hold. This is important because educator beliefs about common misconceptions can diverge significantly from actual student patterns, and traditional approaches rely on predefined taxonomies that cannot capture the full diversity of misconceptions students exhibit. Reasoning Trajectories discussed in Chapter 4 offer a well-defined, principled approach to planning a type of Socratic conversations that are designed to expose contradictions between misconceptions and program behavior through structured proofs-by-counterexample, ensuring dialogues lead to genuine cognitive dissonance and belief revision rather than simply telling students the solution. This is important because existing systems lack a principled framework for systematically planning Socratic conversations, risking questions that are too direct, too vague, or irrelevant.

5.1 Lessons Learned

The process of developing this computational framework revealed several fundamental insights about the nature of Socratic guidance and the challenges inherent to formalizing Socratic debugging as a computational task.

These lessons highlight a fundamental tension in automated Socratic systems: while computational frameworks require precise specifications and evaluation criteria, effective pedagogy inherently involves nuanced, context-dependent decision-making that resists formalization. The multi-reference annotation approach in the Socratic Debugging Benchmark (Chapter 2) represents one attempt to address this challenge by acknowledging that multiple distinct utterances can be pedagogically valid at each conversation turn. Similarly, the reasoning trajectory framework (Chapter 4) narrows the scope to one specific type of Socratic guidance where formalization is more tractable. These design choices reflect a

Lessons Learned.

- **Socratic utterances have distinct connotations and difficulty.** Two utterances may provide equivalent guidance yet differ in their connotations and perceived difficulty. For instance, in Chapter 2 (Section 2.5.1), the utterance "How does the range function work in your loop, and what values does it generate for `i`?" matches "Can you tell me what `range(n)` does?" in terms of guidance, yet the former may be perceived as more specific and concrete, while the latter appears more open-ended, potentially intimidating, and more difficult to answer.
- **Multiple valid guidance paths exist at each conversation point.** At any given turn in a Socratic conversation, instructors can follow multiple distinct approaches to guide students toward discovering bugs and correcting misconceptions (Chapter 2).
- **Misconceptions are diverse and not always harmful.** As demonstrated in Chapter 3, misconceptions can be classified as benign or harmful depending on whether they cause bugs. Many misconceptions require explicit probing through conversation to verify their presence, as they may not manifest as detectable errors in student code.
- **Socratic guidance lacks precise definition in existing literature.** Instructors employ varied approaches including program tracing, state inspection through debugging tools, abstract reasoning about program behavior, and analogical reasoning. The framework presented in Chapter 4 focuses specifically on one well-defined type: guiding students through deductive reasoning steps that lead to statements contradicting their misconceptions.
- **Formalizing Socratic matching is operationally challenging.** Determining whether two instructor utterances are Socratically equivalent requires assessing whether they demand the same knowledge and skills from students and lead to equivalent progress in solving the problem. However, defining "progress" and "effort" operationally remains non-trivial, as these constructs depend on individual student characteristics, prior knowledge, and cognitive states that are difficult to measure or model precisely.

pragmatic approach: rather than attempting to capture all possible forms of Socratic interaction, we focus on well-defined subtypes that can be rigorously evaluated while remaining pedagogically valuable. Future work in this area must continue to balance the need for operational definitions with the inherent flexibility and adaptability that characterize expert human instruction.

Looking forward, these insights inform several promising directions for extending and refining the computational framework presented in this dissertation.

5.2 Future Directions

Several promising directions emerge from this work, spanning extensions of the technical framework, applications to new domains, and integration with educational practice.

▶ **Extending the computational framework.** The current framework focuses on debugging simple computational problems, but can be extended in multiple dimensions. First, it can encompass programming from scratch, where agents guide students through the entire problem-solving process. Second, the framework can be adapted to other programming languages such as Java and C++, each presenting unique misconception patterns and debugging challenges. Third, the Socratic debugging task can be applied to more complex computational problems including parallel computing, object-oriented programming, and data structures, where misconceptions might involve intricate conceptual relationships.

▶ **Generalizing beyond programming.** The Reasoning Trajectory approach presented in Chapter 4 is domain-independent in principle. Any field requiring problem-solving through misconception correction could benefit from this framework, including mathematics (e.g., algebraic reasoning, proof construction), physics, and chemistry. Adapting the framework to these domains would require identifying domain-specific misconception taxonomies and formalizing the structure of valid reasoning trajectories within each field.

▶ **Optimization and personalization.** A critical direction for future research is adapting Socratic dialogue to individual student characteristics and learning contexts. This includes developing methods to dynamically adjust the level of guidance based on student responses and prior knowledge and determining when to provide more direct instruction versus Socratic questioning. Machine learning approaches could be employed to learn student models from interaction data, enabling more effective personalization over time.

► **Classroom integration and evaluation.** While the computational framework has been evaluated on benchmark datasets, real classroom deployments are essential for assessing learning outcomes and pedagogical effectiveness. This includes longitudinal studies measuring knowledge retention and transfer, comparative studies of AI-guided versus human-guided Socratic instruction, and investigation of how Socratic agents can complement rather than replace human teachers in blended learning environments.

► **Scaled misconception mining.** The McMining framework (Chapter 3) demonstrates automated misconception discovery, but could be extended to continuously mine and update misconception taxonomies from large-scale student interaction data. This would enable the construction of comprehensive, data-driven knowledge bases that capture evolving patterns of student difficulties across diverse populations, programming languages, and problem types.

5.3 Broader Impacts

This work contributes to a fundamental reconceptualization of AI's role in education: not as an oracle that provides answers, but as a Socratic partner that enhances human thinking through questioning and guidance. As AI capabilities continue to advance, this distinction becomes increasingly critical to ensure that automation augments rather than diminishes human cognitive development. By establishing computational frameworks for AI systems that foster productive struggle and guided discovery rather than replacing human thinking, this dissertation offers a pathway toward leveraging AI's capabilities while preserving the cognitive exercise essential for genuine learning.

The deployment of AI-based Socratic agents in educational settings raises important considerations regarding privacy, fairness, and pedagogical effectiveness. Students should be informed about the use of AI in their learning process and have agency over their participation. Most importantly, Socratic agents should complement rather than replace human instruction, serving as a scalable supplement to help educators provide more personalized attention to students who need it most.

REFERENCES

[1] S. Papert and I. Harel, “Situating constructionism,”

[2] S. Freeman, S. L. Eddy, M. McDonough, M. K. Smith, N. Okoroafor, H. Jordt, and M. P. Wenderoth, “Active learning increases student performance in science, engineering, and mathematics,” *Proceedings of the National Academy of Sciences*, vol. 111, no. 23, pp. 8410–8415, 2014.

[3] W. Uzgalis, “John locke (stanford encyclopedia of philosophy),” *Stanford Encyclopedia of Philosophy*, vol. 5, 2007.

[4] F. Lam, “The Socratic Method as an Approach to Learning and Its Benefits,” *Carnegie Mellon University*, 1 2011.

[5] J. D. Bransford, A. L. Brown, R. R. Cocking, *et al.*, *How people learn*, vol. 11. Washington, DC: National academy press, 2000.

[6] R. Bommasani, D. A. Hudson, E. Adeli, R. Altman, S. Arora, S. von Arx, M. S. Bernstein, J. Bohg, A. Bosselut, E. Brunskill, *et al.*, “On the opportunities and risks of foundation models,” *arXiv preprint arXiv:2108.07258*, 2021.

[7] J. Moor, *The Turing test: the elusive standard of artificial intelligence*, vol. 30. Springer Science & Business Media, 2003.

[8] J. Weizenbaum *et al.*, “Eliza—a computer program for the study of natural language communication between man and machine,” *Communications of the ACM*, vol. 9, no. 1, pp. 36–45, 1966.

[9] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, \. Kaiser, and I. Polosukhin, “Attention is all you need,” in *Advances in neural information processing systems*, pp. 5998–6008, 2017.

[10] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, *et al.*, “Language models are few-shot learners,” *arXiv preprint arXiv:2005.14165*, 2020.

[11] L. Ouyang, J. Wu, X. Jiang, D. Almeida, C. Wainwright, P. Mishkin, C. Zhang, S. Agarwal, K. Slama, A. Ray, *et al.*, “Training language models to follow instructions with human feedback,” *Advances in Neural Information Processing Systems*, vol. 35, pp. 27730–27744, 2022.

[12] J. L. Z. Montenegro, C. A. da Costa, and R. da Rosa Righi, “Survey of conversational agents in health,” *Expert Systems with Applications*, vol. 129, pp. 56–67, 2019.

[13] S. D’Alfonso, O. Santesteban-Echarri, S. Rice, G. Wadley, R. Lederman, C. Miles, J. Gleeson, and M. Alvarez-Jimenez, “Artificial intelligence-assisted online social therapy for youth mental health,” *Frontiers in psychology*, vol. 8, p. 796, 2017.

[14] S. Hobert, “Say Hello to ‘Coding Tutor’! Design and Evaluation of a Chatbot-based Learning System Supporting Students to Learn to Program,” *ICIS 2019 Proceedings*, Nov. 2019.

[15] Anthropic, “Introducing Claude for Education,” 2024.

[16] OpenAI, “Introducing study mode,” 2025.

[17] Google, “Guided Learning in Gemini: From answers to understanding,” 2025.

[18] A. Modi, A. S. Veerubhotla, A. Rysbek, A. Huber, A. Anand, A. Bhoopchand, B. Wiltshire, D. Gillick, D. Kasenberg, E. Sgouritsa, G. Elidan, H. Liu, H. Winnemoeller, I. Jurenka, J. Cohan, J. She, J. Wilkowski, K. Alarakyia, K. R. McKee, K. Singh, L. Wang, M. Kunesch, M. Pîslar, N. Efron, P. Mahmoudieh, P.-A. Kamienny, S. Wiltberger, S. Mohamed, S. Agarwal, S. M. Phal, S. J. Lee, T. Strinopoulos, W.-J. Ko, Y. Gold-Zamir, Y. Haramaty, Y. Assael, *et al.*, “Evaluating Gemini in an Arena for Learning,” 2025.

[19] M. Wu, N. Goodman, C. Piech, and C. Finn, “Prototransformer: A meta-learning approach to providing student feedback,” *arXiv preprint arXiv:2107.14035*, 2021.

[20] S. MacNeil, A. Tran, J. Leinonen, P. Denny, J. Kim, A. Hellas, S. Bernstein, and S. Sarsa, “Automatically generating cs learning materials with large language models,” in *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 2*, SIGCSE 2023, (New York, NY, USA), p. 1176, Association for Computing Machinery, 2023.

[21] S. Sarsa, P. Denny, A. Hellas, and J. Leinonen, “Automatic generation of programming exercises and code explanations using large language models,” in *Proceedings of the 2022 ACM Conference on International Computing Education Research - Volume 1*, ICER ’22, (New York, NY, USA), p. 27–43, Association for Computing Machinery, 2022.

[22] S. MacNeil, A. Tran, A. Hellas, J. Kim, S. Sarsa, P. Denny, S. Bernstein, and J. Leinonen, “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 V. 1*, SIGCSE 2023, (New York, NY, USA), p. 931–937, Association for Computing Machinery, 2023.

[23] J. Jeuring, L. T. van Binsbergen, A. Gerdes, and B. Heeren, “Model solutions and properties for diagnosing student programs in ask-elle,” in *Proceedings of the Computer Science Education Research Conference*, CSERC ’14, (New York, NY, USA), p. 31–40, Association for Computing Machinery, 2014.

[24] A. Gerdes, B. Heeren, J. Jeuring, and L. T. Van Binsbergen, “Ask-elle: an adaptable programming tutor for haskell giving automated feedback,” *International Journal of Artificial Intelligence in Education*, vol. 27, no. 1, pp. 65–100, 2017.

[25] Z. Alshaikh, L. J. Tamang, and V. Rus, “Experiments with a Socratic intelligent tutoring system for source code understanding,” in *The Thirty-Third International Florida Artificial Intelligence Research Society Conference (FLAIRS-32)*, 2020.

[26] K. Shridhar, J. Macina, M. El-Assady, T. Sinha, M. Kapur, and M. Sachan, “Automatic generation of Socratic subquestions for teaching math word problems,” in *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, (Abu Dhabi, United Arab Emirates), pp. 4136–4149, Association for Computational Linguistics, Dec. 2022.

[27] J. Macina, N. Daheim, S. Chowdhury, T. Sinha, M. Kapur, I. Gurevych, and M. Sachan, “MathDial: A dialogue tutoring dataset with rich pedagogical properties grounded in math reasoning problems,” in *Findings of the Association for Computational Linguistics: EMNLP 2023* (H. Bouamor, J. Pino, and K. Bali, eds.), (Singapore), pp. 5602–5621, Association for Computational Linguistics, Dec. 2023.

[28] J. Macina, N. Daheim, I. Hakimi, M. Kapur, I. Gurevych, and M. Sachan, “Mathtutorbench: A benchmark for measuring open-ended pedagogical capabilities of llm tutors,” *arXiv preprint arXiv:2502.18940*, 2025.

[29] J. Liu, Z. Huang, T. Xiao, J. Sha, J. Wu, Q. Liu, S. Wang, and E. Chen, “Socraticlm: exploring Socratic personalized teaching with large language models,” in *Proceedings of the 38th International Conference on Neural Information Processing Systems*, NIPS ’24, (Red Hook, NY, USA), Curran Associates Inc., 2025.

[30] J. Perczel, J. Chow, and D. Demszky, “Teachlm: Post-training llms for education using authentic learning data,” *arXiv preprint arXiv:2510.05087*, 2025.

[31] J. Wei, M. Bosma, V. Y. Zhao, K. Guu, A. W. Yu, B. Lester, N. Du, A. M. Dai, and Q. V. Le, “Finetuned Language Models Are Zero-Shot Learners,” 2022.

[32] B. Peng, C. Li, P. He, M. Galley, and J. Gao, “Instruction Tuning with GPT-4,” 2023.

[33] Y. Bai, A. Jones, K. Ndousse, A. Askell, A. Chen, N. DasSarma, D. Drain, S. Fort, D. Ganguli, T. Henighan, N. Joseph, S. Kadavath, J. Kernion, T. Conerly, S. El-Showk, N. Elhage, Z. Hatfield-Dodds, D. Hernandez, T. Hume, S. Johnston, S. Kravec, L. Lovitt, N. Nanda, C. Olsson, D. Amodei, T. Brown, J. Clark, S. McCandlish, C. Olah, B. Mann, and J. Kaplan, “Training a Helpful and Harmless Assistant with Reinforcement Learning from Human Feedback,” 2022.

[34] R. A. Bjork and E. L. Bjork, “Desirable Difficulties in Theory and Practice,” *Journal of Applied Research in Memory and Cognition*, vol. 9, no. 4, pp. 475–479, 2020.

[35] J. M. Lodge, G. Kennedy, L. Lockyer, A. Arguel, and M. Pachman, “Understanding Difficulties and Resulting Confusion in Learning: An Integrative Review,” *Frontiers in Education*, vol. 3, pp. 1–10, 2018.

[36] E. L. Bjork and R. A. Bjork, “Making Things Hard on Yourself, But in a Good Way: Creating Desirable Difficulties to Enhance Learning,” in *Psychology and the Real World: Essays Illustrating Fundamental Contributions to Society* (M. A. Gernsbacher, R. W. Pew, L. M. Hough, and J. R. Pomerantz, eds.), pp. 56–64, New York, NY, USA: Worth Publishers, 2011.

[37] P. F. Christiano, J. Leike, T. Brown, M. Martic, S. Legg, and D. Amodei, “Deep Reinforcement Learning from Human Preferences,” *arXiv preprint* , 2017.

[38] D. M. Ziegler, N. Stiennon, J. Wu, T. B. Brown, A. Radford, D. Amodei, P. Christiano, and G. Irving, “Fine-Tuning Language Models from Human Preferences,” 2019.

[39] B. Du Boulay, *Intelligent Tutoring Systems That Adapt to Learner Motivation*, pp. 103–128. Nova Science Publishers Inc, 10 2018.

[40] A. Koriat and R. A. Bjork, “Illusions of competence in monitoring one’s knowledge during study.,” *Journal of experimental psychology. Learning, memory, and cognition*, vol. 31 2, pp. 187–94, 2005.

[41] R. A. Bjork, J. Dunlosky, and N. Kornell, “Self-regulated learning: Beliefs, techniques, and illusions,” *Annual review of psychology*, vol. 64, no. 1, pp. 417–444, 2013.

[42] J. Wang and W. Fan, “The Effect of ChatGPT on Students’ Learning Performance, Learning Perception, and Higher-Order Thinking: Insights from a Meta-Analysis,” *Humanities and Social Sciences Communications*, vol. 12, no. 1, p. 621, 2025.

[43] I. Jurenka, M. Kunesch, K. R. McKee, *et al.*, “Towards Responsible Development of Generative AI for Education: An Evaluation-Driven Approach,” 2024.

[44] J. M. Markel, S. G. Opferman, J. A. Landay, and C. Piech, “Gpteach: Interactive ta training with gpt-based students,” in *Proceedings of the Tenth ACM Conference on Learning @ Scale*, L@S ’23, (New York, NY, USA), p. 226–236, Association for Computing Machinery, 2023.

[45] D. Martynova, J. Macina, N. Daheim, N. Yalcin, X. Zhang, and M. Sachan, “Can LLMs effectively simulate human learners? teachers’ insights from tutoring LLM students,” in *Proceedings of the 20th Workshop on Innovative Use of NLP for Building Educational Applications (BEA 2025)* (E. Kochmar, B. Alhafni, M. Bexte, J. Burstein, A. Horbach, R. Laarmann-Quante, A. Tack, V. Yaneva, and Z. Yuan, eds.), (Vienna, Austria), pp. 100–117, Association for Computational Linguistics, July 2025.

[46] T. Camp, W. R. Adrion, B. Bizot, S. Davidson, M. Hall, S. Hambrusch, E. Walker, and S. Zweben, “Generation cs: the growth of computer science,” *ACM Inroads*, vol. 8, no. 2, pp. 44–50, 2017.

[47] T. Wood, “Patterns of interaction and the culture of mathematics classrooms,” in *Cultural perspectives on the mathematics classroom*, pp. 149–168, Springer, 1994.

[48] D. Wood, J. S. Bruner, and G. Ross, “The role of tutoring in problem solving.,” *Child Psychology & Psychiatry & Allied Disciplines*, 1976.

[49] C. Quintana, B. J. Reiser, E. A. Davis, J. Krajcik, E. Fretz, R. G. Duncan, E. Kyza, D. Edelson, and E. Soloway, “A scaffolding design framework for software to support science inquiry,” in *The journal of the learning sciences*, pp. 337–386, Psychology Press, 2018.

[50] L. S. Vygotsky, *Thought and language*. MIT press, 2012.

[51] D. Mirza, P. T. Conrad, C. Lloyd, Z. Matni, and A. Gatin, “Undergraduate teaching assistants in computer science: a systematic literature review,” in *Proceedings of the 2019 ACM Conference on International Computing Education Research*, pp. 31–40, 2019.

[52] B. S. Bloom, “The 2 sigma problem: The search for methods of group instruction as effective as one-to-one tutoring,” *Educational researcher*, vol. 13, no. 6, pp. 4–16, 1984.

[53] W. G. Bowen, “The ‘cost disease’in higher education: is technology the answer?,” *The Tanner Lectures Stanford University*, 2012.

[54] A. Yadav, S. Gretter, S. Hambrusch, and P. Sands, “Expanding computer science education in schools: understanding teacher experiences and challenges,” *Computer Science Education*, vol. 26, no. 4, pp. 235–254, 2016.

[55] M. L. Maher, C. Latulipe, H. Lipford, and A. Rorrer, “Flipped classroom strategies for CS Education,” in *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*, pp. 218–223, 2015.

[56] N. J. Kim, B. R. Belland, and A. E. Walker, “Effectiveness of computer-based scaffolding in the context of problem-based learning for stem education: Bayesian meta-analysis,” *Educational Psychology Review*, vol. 30, no. 2, pp. 397–429, 2018.

[57] N. Kosmyna, E. Hauptmann, Y. T. Yuan, J. Situ, X.-H. Liao, A. V. Beresnitzky, I. Braunstein, and P. Maes, “Your brain on chatgpt: Accumulation of cognitive debt when using an ai assistant for essay writing task,” *arXiv preprint arXiv:2506.08872*, 2025.

[58] A. Wood, P. Rodeghero, A. Armaly, and C. McMillan, “Detecting speech act types in developer question/answer conversations during bug repair,” in *Proceedings of the 2018 26th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering*, pp. 491–502, 2018.

[59] A. Wood, Z. Eberhart, and C. McMillan, “Dialogue act classification for virtual agents for software engineers during debugging,” in *Proceedings of the IEEE/ACM 42nd international conference on software engineering workshops*, pp. 462–469, 2020.

[60] L. Chen, B. Di Eugenio, D. Fossati, S. Ohlsson, and D. Cosejo, "Exploring effective dialogue act sequences in one-on-one computer science tutoring dialogues," in *Proceedings of the Sixth Workshop on Innovative Use of NLP for Building Educational Applications*, (Portland, Oregon), pp. 65–75, Association for Computational Linguistics, June 2011.

[61] A. Caines, H. Yannakoudakis, H. Edmondson, H. Allen, P. Pérez-Paredes, B. Byrne, and P. Buttery, "The teacher-student chatroom corpus," 2020.

[62] K. Stasaski, K. Kao, and M. A. Hearst, "Cima: A large open access dialogue dataset for tutoring," in *Proceedings of the Fifteenth Workshop on Innovative Use of NLP for Building Educational Applications*, pp. 52–64, 2020.

[63] A. Suresh, J. Jacobs, M. Perkoff, J. H. Martin, and T. Sumner, "Fine-tuning transformers with additional context to classify discursive moves in mathematics classrooms," in *Proceedings of the 17th Workshop on Innovative Use of NLP for Building Educational Applications (BEA 2022)* (E. Kochmar, J. Burstein, A. Horbach, R. Laarmann-Quante, N. Madnani, A. Tack, V. Yaneva, Z. Yuan, and T. Zesch, eds.), (Seattle, Washington), pp. 71–81, Association for Computational Linguistics, July 2022.

[64] D. Demszky and H. Hill, "The NCTE Transcripts: A Dataset of Elementary Math Classroom Transcripts," 2023.

[65] J. Vasselli, C. Vasselli, A. Nohejl, and T. Watanabe, "NAISTeacher: A Prompt and Rerank Approach to Generating Teacher Utterances in Educational Dialogues," in *Proceedings of the 18th Workshop on Innovative Use of NLP for Building Educational Applications (BEA 2023)*, pp. 772–784, 2023.

[66] J. Macina, N. Daheim, L. Wang, T. Sinha, M. Kapur, I. Gurevych, and M. Sachan, "Opportunities and challenges in neural dialog tutoring," 2023.

[67] National Tutoring Observatory, "National Tutoring Observatory," 2025.

[68] L. DiPietro, "National Tutoring Observatory to Accelerate the Science of Teaching," *Cornell Chronicle*.

[69] E. Al-Hossami, R. Bunescu, R. Teehan, L. Powell, K. Mahajan, and M. Dorodchi, "Socratic questioning of novice debuggers: A benchmark dataset and preliminary evaluations," in *Proceedings of the 18th Workshop on Innovative Use of NLP for Building Educational Applications (BEA 2023)* (E. Kochmar, J. Burstein, A. Horbach, R. Laarmann-Quante, N. Madnani, A. Tack, V. Yaneva, Z. Yuan, and T. Zesch, eds.), (Toronto, Canada), pp. 709–726, Association for Computational Linguistics, July 2023.

[70] E. Al-Hossami, R. Bunescu, J. Smith, and R. Teehan, "Can language models employ the Socratic method? experiments with code debugging," in *Proceedings of the 55th ACM Technical Symposium on Computer Science Education V. 1*, SIGCSE 2024, (New York, NY, USA), p. 53–59, Association for Computing Machinery, 2024.

[71] W.-C. Kwan, X. Zeng, Y. Jiang, Y. Wang, L. Li, L. Shang, X. Jiang, Q. Liu, and K.-F. Wong, "MT-Eval: A Multi-Turn Capabilities Evaluation Benchmark for Large Language Models," 2024.

[72] V. Sirdeshmukh, K. Deshpande, J. Mols, L. Jin, E.-Y. Cardona, D. Lee, J. Kritz, W. Primack, S. Yue, and C. Xing, "MultiChallenge: A Realistic Multi-Turn Conversation Evaluation Benchmark Challenging to Frontier LLMs," 2025.

[73] Y. Qian and J. Lehman, "Students' Misconceptions and Other Difficulties in Introductory Programming: A Literature Review," *ACM Trans. Comput. Educ.*, vol. 18, pp. 1:1–1:24, Oct. 2017.

[74] A. Robins, "Learning edge momentum: a new account of outcomes in CS1," *Computer Science Education*, vol. 20, pp. 37–71, Mar. 2010. Publisher: Routledge _eprint: https://doi.org/10.1080/08993401003612167.

[75] N. C. Brown and A. Altadmri, "Investigating novice programming mistakes: educator beliefs vs student data," in *Proceedings of the 2014 conference on Innovation & technology in computer science education*, pp. 43–50, 2014.

[76] J. McBroom, I. Koprinska, and K. Yacef, "A survey of automated programming hint generation: The hints framework," *ACM Computing Surveys (CSUR)*, vol. 54, no. 8, pp. 1–27, 2021.

[77] Z. Alshaikh, L. Tamang, and V. Rus, "A Socratic tutor for source code comprehension," in *Artificial Intelligence in Education: 21st International Conference, AIED 2020, Ifrane, Morocco, July 6–10, 2020, Proceedings, Part II 21*, pp. 15–19, Springer, 2020.

[78] M. Dorodchi, A. Benedict, and E. Al-Hossami, "CS1 Scaffolded Activities: The Rise of Students' Engagement," in *Proceedings of the 2019 ACM Conference on International Computing Education Research*, ICER '19, (Toronto ON, Canada), p. 299, Association for Computing Machinery, July 2019.

[79] M. Dorodchi, N. Dehbozorgi, A. Benedict, E. Al-Hossami, and A. Benedict, "Scaffolding a Team-based Active Learning Course to Engage Students: A Multidimensional Approach," in *2020 ASEE Virtual Annual Conference Content Access*, (Virtual On line), ASEE Conferences, June 2020.

[80] E. Demeter, M. Dorodchi, E. Al-Hossami, A. Benedict, L. Slattery Walker, and J. Smail, "Predicting first-time-in-college students' degree completion outcomes," *Higher Education*, vol. 84, no. 3, pp. 589–609, 2022.

[81] D. Brown, M. Uddin, E. Al-Hossami, D. Janies, S. Shaikh, and Z. Cheng, "Multidisciplinary engagement of diverse students in computer science education through research focused on social media covid-19 misinformation," in *2022 ASEE Annual Conference & Exposition*, no. 10.18260/1-2–41892, (Minneapolis, MN), ASEE Conferences, August 2022. https://peer.asee.org/41892.

[82] P. Liguori, E. Al-Hossami, D. Cotroneo, R. Natella, B. Cukic, and S. Shaikh, "Shellcode_IA32: A dataset for automatic shellcode generation," in *Proceedings of the 1st Workshop on Natural Language Processing for Programming (NLP4Prog 2021)*, (Online), pp. 58–64, Association for Computational Linguistics, Aug. 2021.

[83] P. Liguori, E. Al-Hossami, V. Orbinato, R. Natella, S. Shaikh, D. Cotroneo, and B. Cukic, "Evil: Exploiting software via natural language," in *2021 IEEE 32nd International Symposium on Software Reliability Engineering (ISSRE)*, pp. 321–332, 2021.

[84] Y. Frempong, Y. Snyder, E. Al-Hossami, M. Sridhar, and S. Shaikh., "Hijax: Human intent javascript xss generator," in *Proceedings of the 18th International Conference on Security and Cryptography - SECRYPT,*, pp. 798–805, INSTICC, SciTePress, 2021.

[85] E. Al-Hossami and S. Shaikh, "A survey on artificial intelligence for source code: A dialogue systems perspective," *arXiv preprint arXiv:2202.04847*, 2022.

[86] P. Liguori, E. Al-Hossami, D. Cotroneo, R. Natella, B. Cukic, and S. Shaikh, "Can we generate shellcodes via natural language? an empirical study," *Automated Software Engineering*, vol. 29, no. 1, p. 30, 2022.

[87] E. Al-Hossami and R. Bunescu, "Mcmining: Automated discovery of misconceptions in student code," *arXiv preprint arXiv:2510.08827*, 2025.

[88] E. Al-Hossami and R. Bunescu, "Reasoning trajectories for Socratic debugging of student code: From misconceptions to contradictions and updated beliefs," *arXiv preprint arXiv:2511.00371*, 2025.

[89] J. D. Wilson, "A Socratic approach to helping novice programmers debug programs," *ACM SIGCSE Bulletin*, vol. 19, no. 1, pp. 179–182, 1987.

[90] D. N. Perkins and F. Martin, "Fragile knowledge and neglected strategies in novice programmers," in *Papers presented at the first workshop on empirical studies of programmers on Empirical studies of programmers*, pp. 213–229, 1986.

[91] M. Yasunaga and P. Liang, "Break-it-fix-it: Unsupervised learning for program repair," in *International Conference on Machine Learning*, pp. 11941–11952, PMLR, 2021.

[92] A. Ettles, A. Luxton-Reilly, and P. Denny, "Common logic errors made by novice programmers," in *Proceedings of the 20th Australasian Computing Education Conference*, ACE '18, (New York, NY, USA), pp. 83–89, Association for Computing Machinery, Jan. 2018.

[93] Y. Hu, U. Z. Ahmed, S. Mechtaev, B. Leong, and A. Roychoudhury, "Re-factoring based program repair applied to programming assignments," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 388–398, IEEE/ACM, 2019.

[94] A. de Freitas, J. Coffman, M. de Freitas, J. Wilson, and T. Weingart, "FalconCode: A Multiyear Dataset of Python Code Samples from an Introductory Computer Science Course," in *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1*, SIGCSE 2023, (New York, NY, USA), pp. 938–944, Association for Computing Machinery, Mar. 2023.

[95] J. C. Spohrer, E. Pope, M. Lipman, W. Sack, S. Freiman, D. Littman, W. Johnson, and E. Soloway, "Bug catalogue: Ii, iii, iv," *Yale University, Department of Computer Science*, 1985.

[96] D. N. Perkins and F. Martin, "Fragile knowledge and neglected strategies in novice programmers," in *Papers Presented at the First Workshop on Empirical Studies of Programmers on Empirical Studies of Programmers*, (USA), p. 213–229, Ablex Publishing Corp., 1986.

[97] R. McCauley, S. Fitzgerald, G. Lewandowski, L. Murphy, B. Simon, L. Thomas, and C. Zander, "Debugging: a review of the literature from an educational perspective," *Computer Science Education*, June 2008. Publisher: Routledge.

[98] P. Gupta, S. Mehri, T. Zhao, A. Pavel, M. Eskenazi, and J. P. Bigham, "Investigating evaluation of open-domain dialogue systems with human generated multiple references," in *Proceedings of the 20th Annual SIGdial Meeting on Discourse and Dialogue*, pp. 379–391, 2019.

[99] OpenAI, "Introducing chatgpt," 2022.

[100] OpenAI, "Gpt-4 technical report," *arXiv*, 2023.

[101] OpenAI, "GPT-4 Blog." https://openai.com/research/gpt-4, 2023. [Online; accessed 19-August-2023].

[102] J. Wei, X. Wang, D. Schuurmans, M. Bosma, F. Xia, E. Chi, Q. V. Le, D. Zhou, *et al.*, "Chain-of-thought prompting elicits reasoning in large language models," *Advances in Neural Information Processing Systems*, vol. 35, pp. 24824–24837, 2022.

[103] H. W. Chung, L. Hou, S. Longpre, B. Zoph, Y. Tay, W. Fedus, E. Li, X. Wang, M. Dehghani, S. Brahma, *et al.*, "Scaling instruction-finetuned language models," *arXiv preprint arXiv:2210.11416*, 2022.

[104] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[105] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, "Bleu: a method for automatic evaluation of machine translation," in *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*, (Philadelphia, Pennsylvania, USA), pp. 311–318, Association for Computational Linguistics, July 2002.

[106] T. Zhang, V. Kishore, F. Wu, K. Q. Weinberger, and Y. Artzi, "Bertscore: Evaluating text generation with bert," in *International Conference on Learning Representations*, 2020.

[107] P. He, X. Liu, J. Gao, and W. Chen, "Deberta: Decoding-enhanced bert with disentangled attention," *arXiv preprint arXiv:2006.03654*, 2020.

[108] C.-Y. Lin, "ROUGE: A package for automatic evaluation of summaries," in *Text Summarization Branches Out*, (Barcelona, Spain), pp. 74–81, Association for Computational Linguistics, July 2004.

[109] Z. Galil, "Efficient algorithms for finding maximum matching in graphs," *ACM Computing Surveys (CSUR)*, vol. 18, no. 1, pp. 23–38, 1986.

[110] E. Saule, "Experiences on teaching parallel and distributed computing for undergraduates," in *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pp. 361–368, 2018.

[111] M. M. Dorodchi, N. Dehbozorgi, A. Benedict, E. Al-Hossami, and A. Benedict, "Scaffolding a team-based active learning course to engage students: A multidimensional approach," in *2020 ASEE Virtual Annual Conference Content Access*, 2020.

[112] B. J. Reiser, "Scaffolding complex learning: The mechanisms of structuring and problematizing student work," *The Journal of the Learning sciences*, vol. 13, no. 3, pp. 273–304, 2004.

[113] National Council of Teachers of Mathematics, *Principles to actions: Ensuring mathematical success for all*. Reston, VA: NCTM, National Council of Teachers of Mathematics, 2014.

[114] S. Alic, D. Demszky, Z. Mancenido, J. Liu, H. Hill, and D. Jurafsky, "Computationally identifying funneling and focusing questions in classroom discourse," *BEA 2022*, p. 224, 2022.

[115] T. Lehtinen, A. Lukkarinen, and L. Haaranen, "Students struggle to explain their own program code," in *Proceedings of the 26th ACM Conference on Innovation and Technology in Computer Science Education V. 1*, pp. 206–212, 2021.

[116] L. J. Tamang, Z. Alshaikh, N. A. Khayi, P. Oli, and V. Rus, "A comparative study of free self-explanations and Socratic tutoring explanations for source code comprehension," in *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education*, SIGCSE '21, (New York, NY, USA), pp. 219–225, Association for Computing Machinery, 2021.

[117] T. Crow, A. Luxton-Reilly, and B. Wuensche, "Intelligent tutoring systems for programming education: a systematic review," in *Proceedings of the 20th Australasian Computing Education Conference*, pp. 53–62, 2018.

[118] E. Mousavinasab, N. Zarifsanaiey, S. R. Niakan Kalhori, M. Rakhshan, L. Keikha, and M. Ghazi Saeedi, “Intelligent tutoring systems: a systematic review of characteristics, applications, and evaluation methods,” *Interactive Learning Environments*, vol. 29, no. 1, pp. 142–163, 2021.

[119] R. Costello, *Adaptive intelligent personalised learning (aipl) environment*. PhD thesis, University of Hull, 2012. Copyright - Database copyright ProQuest LLC; ProQuest does not claim copyright in the individual underlying works; Analyte descriptor - Bibliographic data provided by EThOS, the British Library’s UK thesis service: https://ethos.bl.uk/OrderDetails.do?uin=uk.bl.ethos.561018; Last updated - 2021-11-30.

[120] C. J. Butz, S. Hua, and R. B. Maguire, “A web-based bayesian intelligent tutoring system for computer programming,” *Web Intelligence and Agent Systems: An International Journal*, vol. 4, no. 1, pp. 77–97, 2006.

[121] K. Chrysafiadi and M. Virvou, “Evaluating the integration of fuzzy logic into the student model of a web-based learning environment,” *Expert systems with applications*, vol. 39, no. 18, pp. 13127–13134, 2012.

[122] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu, “Exploring the limits of transfer learning with a unified text-to-text transformer,” *Journal of Machine Learning Research*, vol. 21, pp. 1–67, 2020.

[123] G. Tyen, M. Brenchley, A. Caines, and P. Buttery, “Towards an open-domain chatbot for language practice,” in *Proceedings of the 17th Workshop on Innovative Use of NLP for Building Educational Applications (BEA 2022)*, (Seattle, Washington), pp. 234–249, Association for Computational Linguistics, July 2022.

[124] V. C. Lee, Y.-T. Yu, C. M. Tang, T.-L. Wong, and C. K. Poon, “Vida: A virtual debugging advisor for supporting learning in computer programming courses,” *Journal of Computer Assisted Learning*, vol. 34, no. 3, pp. 243–258, 2018.

[125] B. P. Iii, A. Hicks, and T. Barnes, “Generating hints for programming problems using intermediate output,” in *Educational Data Mining 2014*, Citeseer, 2014.

[126] T. Lazar, M. Možina, and I. Bratko, “Automatic extraction of ast patterns for debugging student programs,” in *Artificial Intelligence in Education: 18th International Conference, AIED 2017, Wuhan, China, June 28–July 1, 2017, Proceedings 18*, pp. 162–174, Springer, 2017.

[127] K. Rivers and K. R. Koedinger, “Data-driven hint generation in vast solution spaces: a self-improving python programming tutor,” *International Journal of Artificial Intelligence in Education*, vol. 27, no. 1, pp. 37–64, 2017.

[128] J. Leinonen, A. Hellas, S. Sarsa, B. Reeves, P. Denny, J. Prather, and B. A. Becker, “Using large language models to enhance programming error messages,” in *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V.*

1, SIGCSE 2023, (New York, NY, USA), p. 563–569, Association for Computing Machinery, 2023.

[129] R. McCauley, S. Fitzgerald, G. Lewandowski, L. Murphy, B. Simon, L. Thomas, and C. Zander, "Debugging: a review of the literature from an educational perspective," *Computer Science Education*, vol. 18, no. 2, pp. 67–92, 2008.

[130] R. D. Pea, "Language-independent conceptual "bugs" in novice programming," *Journal of educational computing research*, vol. 2, no. 1, pp. 25–36, 1986.

[131] Y. Qian and J. Lehman, "Students' misconceptions and other difficulties in introductory programming: A literature review," *ACM Transactions on Computing Education (TOCE)*, vol. 18, no. 1, pp. 1–24, 2017.

[132] A. Caines, H. Yannakoudakis, H. Edmondson, H. Allen, P. Pérez-Paredes, B. Byrne, and P. Buttery, "The teacher-student chatroom corpus," in *Proceedings of the 9th Workshop on NLP for Computer Assisted Language Learning*, (Gothenburg, Sweden), pp. 10–20, LiU Electronic Press, Nov. 2020.

[133] A. Caines, H. Yannakoudakis, H. Allen, P. Pérez-Paredes, B. Byrne, and P. Buttery, "The teacher-student chatroom corpus version 2: more lessons, new annotation, automatic detection of sequence shifts," in *Proceedings of the 11th Workshop on NLP for Computer Assisted Language Learning*, (Louvain-la-Neuve, Belgium), pp. 23–35, LiU Electronic Press, Dec. 2022.

[134] S. Walsh, *Investigating classroom discourse*. Routledge, 2006.

[135] D. Demszky, J. Liu, Z. Mancenido, J. Cohen, H. Hill, D. Jurafsky, and T. Hashimoto, "Measuring conversational uptake: A case study on student-teacher interactions," in *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, (Online), pp. 1638–1653, Association for Computational Linguistics, Aug. 2021.

[136] M. Alizadeh, B. Di Eugenio, R. Harsley, N. Green, D. Fossati, and O. AlZoubi, "A study of analogy in computer science tutorial dialogues," *Trees*, vol. 53, no. 19.2, pp. 1–6, 2015.

[137] A. Tack and C. Piech, "The AI teacher test: Measuring the pedagogical ability of blender and GPT-3 in educational dialogues," in *Proceedings of the 15th International Conference on Educational Data Mining* (A. Mitrovic and N. Bosch, eds.), (Durham, United Kingdom), pp. 522–529, International Educational Data Mining Society, July 2022.

[138] S. Roller, E. Dinan, N. Goyal, D. Ju, M. Williamson, Y. Liu, J. Xu, M. Ott, E. M. Smith, Y.-L. Boureau, and J. Weston, "Recipes for building an open-domain chatbot," in *Proceedings of the 16th Conference of the European Chapter of the Association for Computational Linguistics: Main Volume*, (Online), pp. 300–325, Association for Computational Linguistics, Apr. 2021.

[139] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. Ponde, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, *et al.*, "Evaluating large language models trained on code," *arXiv preprint arXiv:2107.03374*, 2021.

[140] P. Liguori, E. Al-Hossami, D. Cotroneo, R. Natella, B. Cukic, and S. Shaikh, "Shellcode_IA32: A dataset for automatic shellcode generation," in *Proc. of the 1st Workshop on Natural Language Processing for Prog. (NLP4Prog 2021)*, (Online), pp. 58–64, Aug. 2021.

[141] R. Gupta, S. Pal, A. Kanade, and S. Shevade, "Deepfix: Fixing common c language errors by deep learning," in *Proceedings of the aaai conference on artificial intelligence*, vol. 31, 2017.

[142] M. Yasunaga and P. Liang, "Graph-based, self-supervised program repair from diagnostic feedback," in *International Conference on Machine Learning (ICML)*, 2020.

[143] T. Lutellier, H. V. Pham, L. Pang, Y. Li, M. Wei, and L. Tan, "CoCoNuT: combining context-aware neural translation models using ensemble for program repair," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2020, (New York, NY, USA), pp. 101–114, Association for Computing Machinery, July 2020.

[144] N. Jiang, T. Lutellier, and L. Tan, "CURE: Code-Aware Neural Machine Translation for Automatic Program Repair," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pp. 1161–1173, May 2021. ISSN: 1558-1225.

[145] H. Hajipour, A. Bhattacharyya, C.-A. Staicu, and M. Fritz, "Samplefix: learning to generate functionally diverse fixes," in *Machine Learning and Principles and Practice of Knowledge Discovery in Databases: International Workshops of ECML PKDD 2021, Virtual Event, September 13-17, 2021, Proceedings, Part II*, pp. 119–133, Springer, 2022.

[146] J. Jung, L. Qin, S. Welleck, F. Brahman, C. Bhagavatula, R. Le Bras, and Y. Choi, "Maieutic prompting: Logically consistent reasoning with recursive explanations," in *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, (Abu Dhabi, United Arab Emirates), pp. 1266–1279, Association for Computational Linguistics, Dec. 2022.

[147] A. Pagnoni, A. R. Fabbri, W. Kryściński, and C.-S. Wu, "Socratic pretraining: Question-driven pretraining for controllable summarization," *arXiv preprint arXiv:2212.10449*, 2022.

[148] L. Chiodini, I. Moreno Santos, A. Gallidabino, A. Tafliovich, A. L. Santos, and M. Hauswirth, "A curated inventory of programming language misconceptions," in *Proceedings of the 26th ACM Conference on Innovation and Technology in Computer Science Education V. 1*, ITiCSE '21, (New York, NY, USA), p. 380–386, Association for Computing Machinery, 2021.

[149] A. Evans, Z. Wang, J. Liu, and M. Zheng, “Side-lib: A library for detecting symptoms of python programming misconceptions,” in *Proceedings of the 2023 Conference on Innovation and Technology in Computer Science Education V. 1*, ITiCSE 2023, (New York, NY, USA), p. 159–165, Association for Computing Machinery, 2023.

[150] J. Austin, A. Odena, M. Nye, M. Bosma, H. Michalewski, D. Dohan, E. Jiang, C. Cai, M. Terry, Q. Le, *et al.*, “Program synthesis with large language models,” *arXiv preprint arXiv:2108.07732*, 2021.

[151] A. Ettles, A. Luxton-Reilly, and P. Denny, “Common logic errors made by novice programmers,” in *Proceedings of the 20th Australasian Computing Education Conference*, ACE ’18, (New York, NY, USA), p. 83–89, Association for Computing Machinery, 2018.

[152] A. de Freitas, J. Coffman, M. de Freitas, J. Wilson, and T. Weingart, “Falconcode: A multiyear dataset of python code samples from an introductory computer science course,” in *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1*, SIGCSE 2023, (New York, NY, USA), p. 938–944, Association for Computing Machinery, 2023.

[153] W. L. Johnson and E. Soloway, “Proust: An automatic debugger for pascal programs,” in *Proceedings of the National Conference on Artificial Intelligence*, pp. 181–184, 1984.

[154] T. Sirkia and J. Sorva, “Exploring programming misconceptions: An analysis of student mistakes in visual program simulation exercises,” in *Proceedings of the 12th Koli Calling International Conference on Computing Education Research*, pp. 19–28, 2012.

[155] Y. Lee, S. Jeong, and J. Kim, “Improving LLM Classification of Logical Errors by Integrating Error Relationship into Prompts,” in *Generative Intelligence and Intelligent Tutoring Systems: 20th International Conference, ITS 2024, Thessaloniki, Greece, June 10–13, 2024, Proceedings, Part I*, (Berlin, Heidelberg), pp. 91–103, Springer-Verlag, June 2024.

[156] K. Mens, S. Nijssen, and H. Pham, “The good, the bad, and the ugly: Mining for patterns in student source code,” in *Proceedings of the 3rd International Workshop on Education through Advanced Software Engineering and Artificial Intelligence (EASEAI ’21)* (A. Vescan, C. Serban, J. Henry, and U. Praphamontripong, eds.), pp. 1–8, ACM, 2021.

[157] M. Hoq, A. Rao, R. Jaishankar, K. Piryani, N. Janapati, J. Vandenberg, B. Mott, N. Norouzi, J. Lester, and B. Akram, “Automated identification of logical errors in programs: Advancing scalable analysis of student misconceptions,” in *Proceedings of the 18th International Conference on Educational Data Mining* (C. Mills, G. Alexandron, D. Taibi, G. L. Bosco, and L. Paquette, eds.), (Palermo, Italy), pp. 90–103, International Educational Data Mining Society, July 2025.

[158] D. Liu, J. Calver, and M. Craig, "A static analysis tool in cs1: Student usage and perceptions of pythonta," in *Proceedings of the 26th Australasian Computing Education Conference*, ACE '24, (New York, NY, USA), pp. 172–181, Association for Computing Machinery, 2024.

[159] L. Elder and R. Paul, "The Role of Socratic Questioning in Thinking, Teaching, and Learning," *The Clearing House: A Journal of Educational Strategies, Issues and Ideas*, vol. 71, pp. 297–301, May 1998.

[160] N. C. C. Brown, F. F. J. Hermans, and L. E. Margulieux, "10 things software developers should learn about learning," *Commun. ACM*, vol. 67, p. 78–87, Dec. 2023.

[161] S. Santhanam, Z. Cheng, B. Mather, B. Dorr, A. Bhatia, B. Hebenstreit, A. Zemel, A. Dalton, T. Strzalkowski, and S. Shaikh, "Learning to plan and realize separately for open-ended dialogue systems," in *Findings of the Association for Computational Linguistics: EMNLP 2020* (T. Cohn, Y. He, and Y. Liu, eds.), (Online), pp. 2736–2750, Association for Computational Linguistics, Nov. 2020.

[162] L. Festinger, "Cognitive dissonance," *New York*, 1959.

[163] M. P. Zanna and J. Cooper, "Dissonance and the attribution process," *New directions in attribution research*, vol. 1, pp. 199–217, 1976.

[164] A. J. Elliot and P. G. Devine, "On the motivational nature of cognitive dissonance: Dissonance as psychological discomfort.," *Journal of personality and social psychology*, vol. 67, no. 3, p. 382, 1994.

[165] A. Adcock, *Cognitive Dissonance in the Learning Processes*, pp. 588–590. Boston, MA: Springer US, 2012.

[166] J. Piaget, T. Brown, *et al.*, "The equilibration of cognitive structures: The central problem of intellectual development," *(No Title)*, 1985.

[167] M. Patel, R. Bhattacharyya, T. Lu, A. Mehta, N. Voss, N. Norouzi, and G. Ranade, "Leantutor: A formally-verified ai tutor for mathematical proofs," *arXiv preprint arXiv:2506.08321*, 2025.

[168] K.-C. Lu and S. Krishnamurthi, "Identifying and correcting programming language behavior misconceptions," *Proc. ACM Program. Lang.*, vol. 8, Apr. 2024.

APPENDIX A: Socratic Debugging Benchmark Appendix

This appendix contains all the prompts, implementation details, and annotation tool for the Socratic Debugging benchmark Chapter 2.

A.1 Data Contribution Web Application

We developed a 7-page data contribution web application tool using the Streamlit Python library[1] to collect dialogues and code snapshots. The application loads a repository of programming problems and bugs from a Google Spreadsheet using the Google Spreadsheet API through the gsheetsdb Python library[2]. The web app consists of the following pages:

- **Getting Started:** This page (Figure A.1) orients the users on the task and provides a link to the guidelines document.
- **Browse Bugs:** Contributors browse and select a bug (Figure A.2) to create a Socratic dialogue for.
- **4 Data Contribution Pages:** These pages contain a code editor and a chat area (Figure A.3) where contributors create an initial conversation and up to 3 conversational threads.
- **Review and Submit:** This page (Figure A.7) allows contributors to review their work and submit the exported dialogues for review.

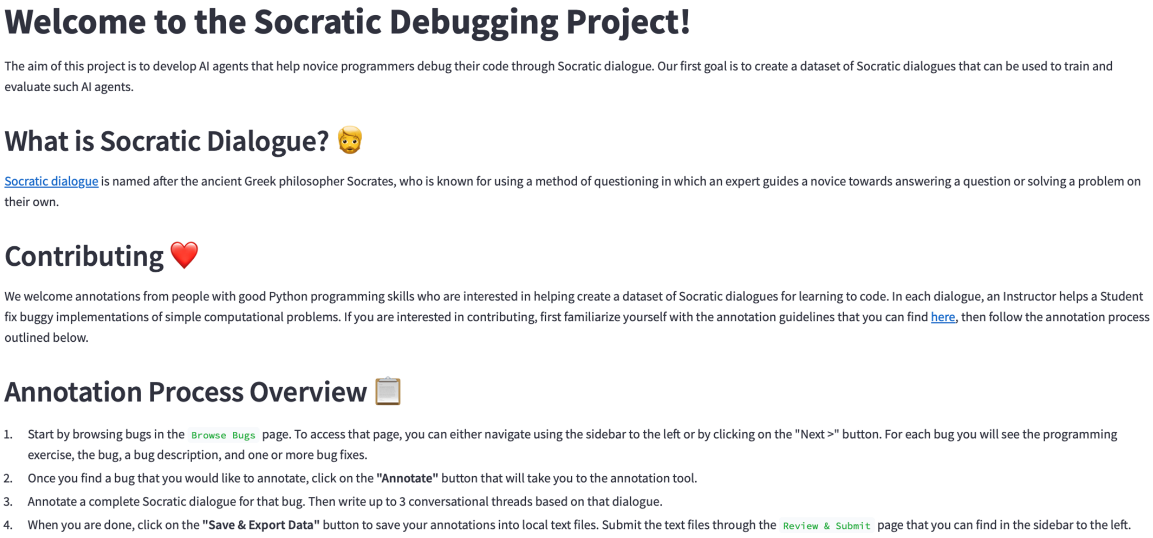
During the data contribution process, contributors can add main and alternative utterances, undo added utterances or code snapshots, and edit the chat history text area and code in

---

[1] https://streamlit.io/

[2] https://github.com/betodealmeida/gsheets-db-api

the code editor. When the contributor edits the code in the Code Editor, they can choose to compile and run the code within the web application and they can also add a code snapshot to the chat history by clicking the "Add Code to Chat History" button (Figure A.4). Once the bug has been fixed, the contributor compiles and runs the code in the Code Editor, as demonstrated in Figure A.5. Contributors can then use the import and export buttons shown in Figure A.6 to save their work. The export button generates a standardized form of the dialogue and code states, while the import button allows contributors to load previously exported dialogues back into the tool. After completing their data contribution, contributors submit the exported dialogues for review.



Welcome to the Socratic Debugging Project!

The aim of this project is to develop AI agents that help novice programmers debug their code through Socratic dialogue. Our first goal is to create a dataset of Socratic dialogues that can be used to train and evaluate such AI agents.

What is Socratic Dialogue? 👱

Socratic dialogue is named after the ancient Greek philosopher Socrates, who is known for using a method of questioning in which an expert guides a novice towards answering a question or solving a problem on their own.

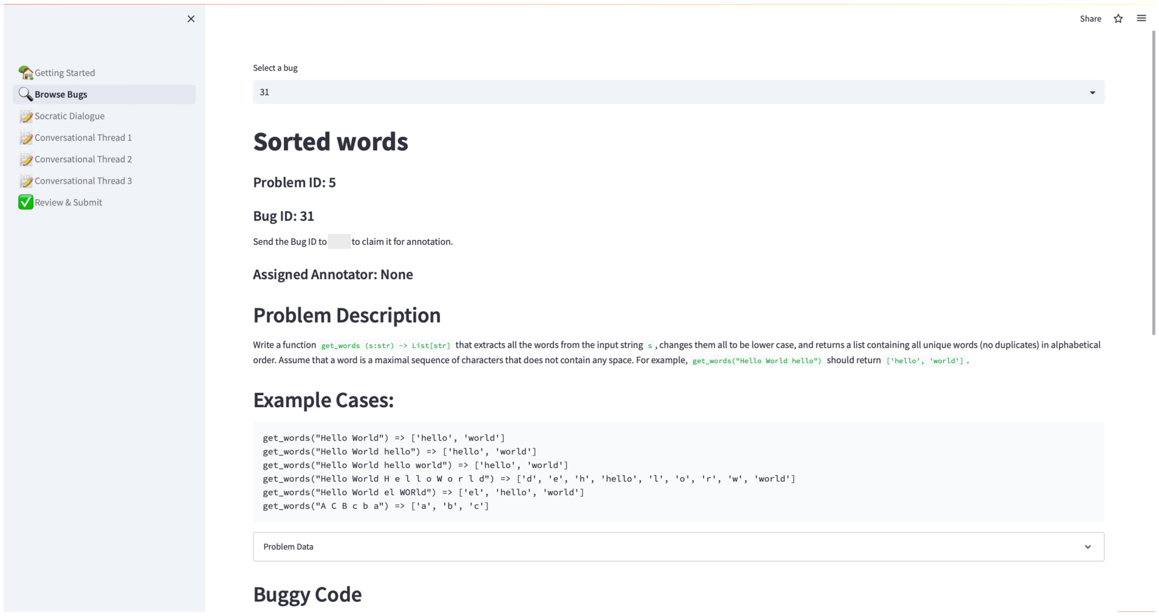
Contributing ❤️

We welcome annotations from people with good Python programming skills who are interested in helping create a dataset of Socratic dialogues for learning to code. In each dialogue, an Instructor helps a Student fix buggy implementations of simple computational problems. If you are interested in contributing, first familiarize yourself with the annotation guidelines that you can find here, then follow the annotation process outlined below.

Annotation Process Overview 📋

1. Start by browsing bugs in the `Browse Bugs` page. To access that page, you can either navigate using the sidebar to the left or by clicking on the "Next >" button. For each bug you will see the programming exercise, the bug, a bug description, and one or more bug fixes.
2. Once you find a bug that you would like to annotate, click on the **"Annotate"** button that will take you to the annotation tool.
3. Annotate a complete Socratic dialogue for that bug. Then write up to 3 conversational threads based on that dialogue.
4. When you are done, click on the **"Save & Export Data"** button to save your annotations into local text files. Submit the text files through the `Review & Submit` page that you can find in the sidebar to the left.

Figure A.1: Screenshot of the web application's Getting Started page where contributors get familiarized with the task and go through the guidelines document.



Select a bug

31

Sorted words

Problem ID: 5

Bug ID: 31

Send the Bug ID to [redacted] to claim it for annotation.

Assigned Annotator: None

Problem Description

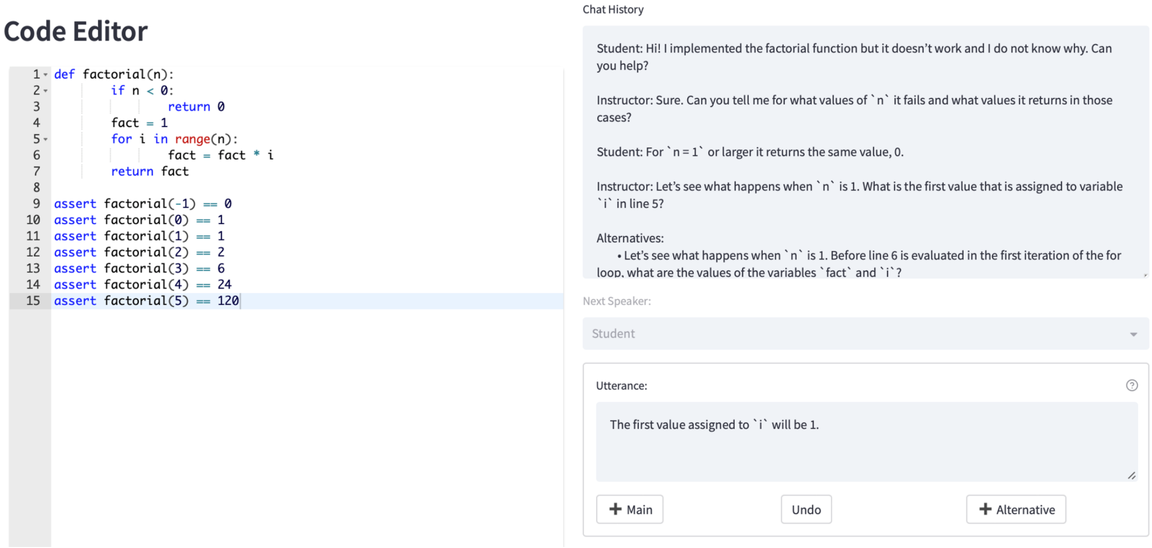
Write a function `get_words (s:str) -> List[str]` that extracts all the words from the input string `s`, changes them all to be lower case, and returns a list containing all unique words (no duplicates) in alphabetical order. Assume that a word is a maximal sequence of characters that does not contain any space. For example, `get_words("Hello World hello")` should return `['hello', 'world']`.

Example Cases:

```
get_words("Hello World") => ['hello', 'world']
get_words("Hello World hello") => ['hello', 'world']
get_words("Hello World hello world") => ['hello', 'world']
get_words("Hello World H e l l o W o r l d") => ['d', 'e', 'h', 'hello', 'l', 'o', 'r', 'w', 'world']
get_words("Hello World el WORld") => ['el', 'hello', 'world']
get_words("A C B c b a") => ['a', 'b', 'c']
```

Buggy Code

Figure A.2: Screenshot of the interface. Contributors first browse a repository of bugs created from a set of programming problems. Each bug is displayed with the problem description, test cases, a buggy code, the bug description, and bug fixes. Contributors select a bug to create a dialogue for.



```
1  def factorial(n):
2          if n < 0:
3                  return 0
4          fact = 1
5          for i in range(n):
6                  fact = fact * i
7          return fact
8
9  assert factorial(-1) == 0
10 assert factorial(0) == 1
11 assert factorial(1) == 1
12 assert factorial(2) == 2
13 assert factorial(3) == 6
14 assert factorial(4) == 24
15 assert factorial(5) == 120
```

Student: Hi! I implemented the factorial function but it doesn't work and I do not know why. Can you help?

Instructor: Sure. Can you tell me for what values of `n` it fails and what values it returns in those cases?

Student: For `n = 1` or larger it returns the same value, 0.

Instructor: Let's see what happens when `n` is 1. What is the first value that is assigned to variable `i` in line 5?

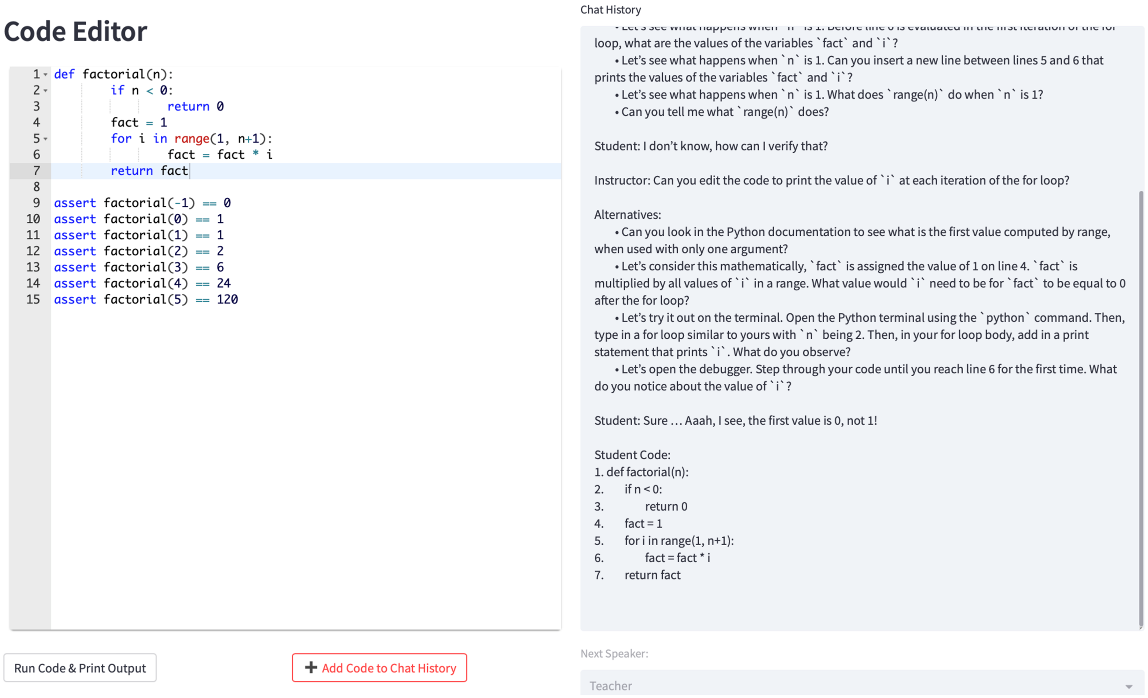
Alternatives:

- Let's see what happens when `n` is 1. Before line 6 is evaluated in the first iteration of the for loop, what are the values of the variables `fact` and `i`?

Student

The first value assigned to `i` will be 1.

Figure A.3: Screenshot of the tool used to collect dialogues and code snapshots. Contributors are able to add a main utterance, an alternative utterance, and undo an adding utterance or a code snapshot. Additionally, the chat history text area is editable.



```
 1 def factorial(n):
 2         if n < 0:
 3                 return 0
 4         fact = 1
 5         for i in range(1, n+1):
 6                 fact = fact * i
 7         return fact
 8
 9 assert factorial(-1) == 0
10 assert factorial(0) == 1
11 assert factorial(1) == 1
12 assert factorial(2) == 2
13 assert factorial(3) == 6
14 assert factorial(4) == 24
15 assert factorial(5) == 120
```

loop, what are the values of the variables `fact` and `i`?
- Let's see what happens when `n` is 1. Can you insert a new line between lines 5 and 6 that prints the values of the variables `fact` and `i`?
- Let's see what happens when `n` is 1. What does `range(n)` do when `n` is 1?
- Can you tell me what `range(n)` does?

Student: I don't know, how can I verify that?

Instructor: Can you edit the code to print the value of `i` at each iteration of the for loop?

Alternatives:
- Can you look in the Python documentation to see what is the first value computed by range, when used with only one argument?
- Let's consider this mathematically, `fact` is assigned the value of 1 on line 4. `fact` is multiplied by all values of `i` in a range. What value would `i` need to be for `fact` to be equal to 0 after the for loop?
- Let's try it out on the terminal. Open the Python terminal using the `python` command. Then, type in a for loop similar to yours with `n` being 2. Then, in your for loop body, add in a print statement that prints `i`. What do you observe?
- Let's open the debugger. Step through your code until you reach line 6 for the first time. What do you notice about the value of `i`?

Student: Sure … Aaah, I see, the first value is 0, not 1!

Student Code:
```
1. def factorial(n):
2.      if n < 0:
3.           return 0
4.      fact = 1
5.      for i in range(1, n+1):
6.           fact = fact * i
7.      return fact
```

Next Speaker:
Teacher

Figure A.4: Screenshot of the tool adding a code snapshot by clicking the Add Code to Chat History button.

```python
def factorial(n):
        if n < 0:
                return 0
        fact = 1
        for i in range(1, n+1):
                fact = fact * i
        return fact

assert factorial(-1) == 0
assert factorial(0) == 1
assert factorial(1) == 1
assert factorial(2) == 2
assert factorial(3) == 6
assert factorial(4) == 24
assert factorial(5) == 120
```

```
Program passes all unit tests!
```

Figure A.5: Screenshot of the tool compiling and running the code in the Code Editor after the bug has been fixed.

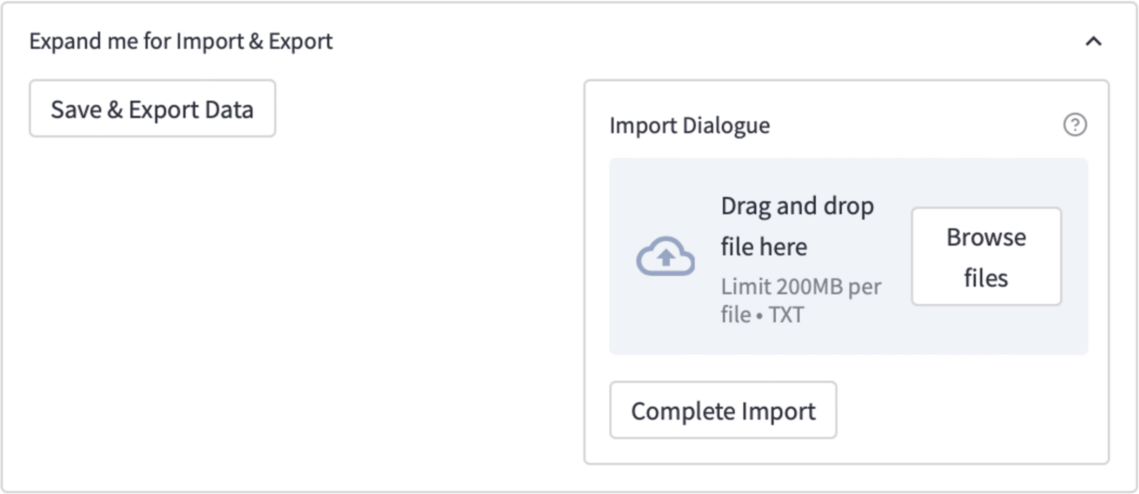
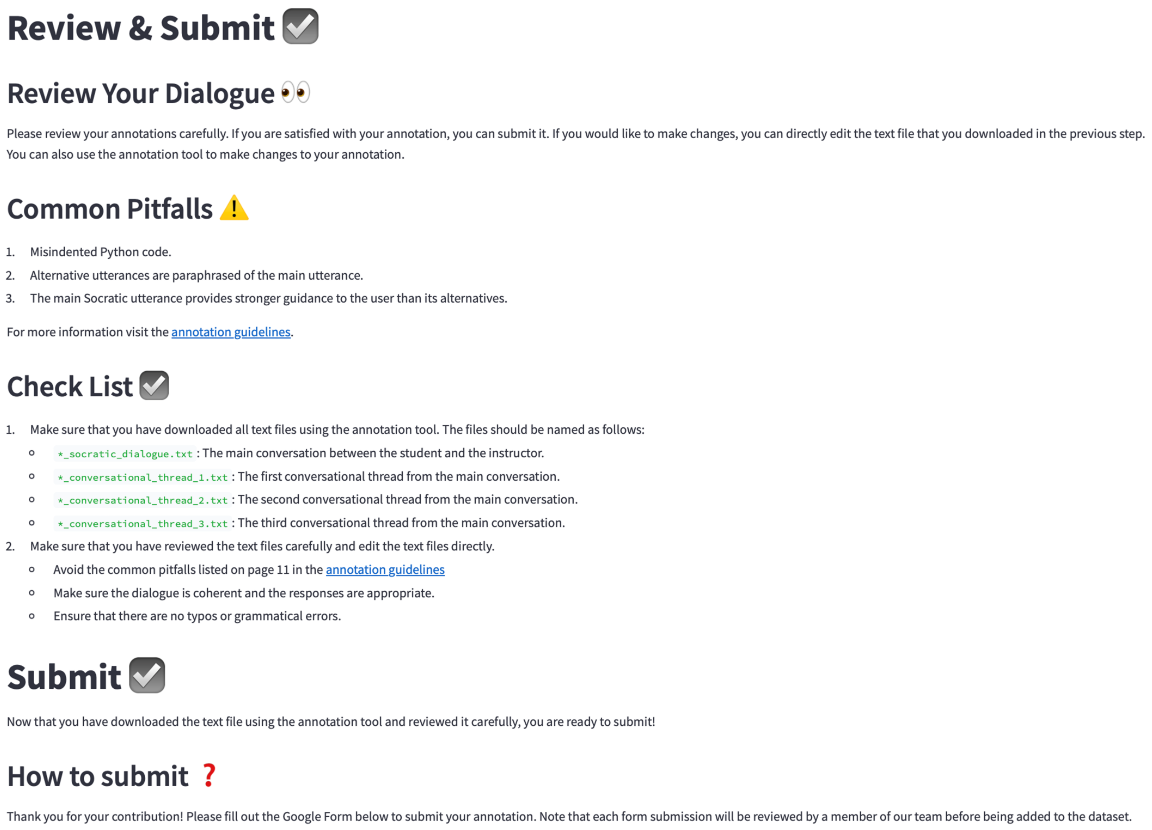


Figure A.6: Screenshot of the tool's import and export buttons. Upon completing a dialogue contributors use the export button to export the dialogue and code states into a standardized form. Additionally, contributors can import any dialogue exported from this tool using the import button.



Review & Submit ☑️

Review Your Dialogue 👀

Please review your annotations carefully. If you are satisfied with your annotation, you can submit it. If you would like to make changes, you can directly edit the text file that you downloaded in the previous step. You can also use the annotation tool to make changes to your annotation.

Common Pitfalls ⚠️

1. Misindented Python code.
2. Alternative utterances are paraphrased of the main utterance.
3. The main Socratic utterance provides stronger guidance to the user than its alternatives.

For more information visit the annotation guidelines.

Check List ☑️

1. Make sure that you have downloaded all text files using the annotation tool. The files should be named as follows:
   - `*_socratic_dialogue.txt` : The main conversation between the student and the instructor.
   - `*_conversational_thread_1.txt` : The first conversational thread from the main conversation.
   - `*_conversational_thread_2.txt` : The second conversational thread from the main conversation.
   - `*_conversational_thread_3.txt` : The third conversational thread from the main conversation.
2. Make sure that you have reviewed the text files carefully and edit the text files directly.
   - Avoid the common pitfalls listed on page 11 in the annotation guidelines
   - Make sure the dialogue is coherent and the responses are appropriate.
   - Ensure that there are no typos or grammatical errors.

Submit ☑️

Now that you have downloaded the text file using the annotation tool and reviewed it carefully, you are ready to submit!

How to submit ❓

Thank you for your contribution! Please fill out the Google Form below to submit your annotation. Note that each form submission will be reviewed by a member of our team before being added to the dataset.

Figure A.7: Screenshot of the web application's Review & Submit page where contributors are instructed to review their data contribution and submit their exported version.

A.2 Language Model Prompt

This section describes the prompt template that was used for language models in this paper. {{text}} denotes a data point from the benchmark dataset. The steering prompt was adapted from the GPT-4 blog post[3]. The '1.' is added at the end of the instruction to prompt the language model to generate an itemized list of utterances that can then be parsed.

Steering Prompt:

> You are a tutor that always responds in the Socratic style. You *never* give the student the answer, but always try to ask just the right question to help them learn to think for themselves. You should always tune your question to the interest & knowledge of the student, breaking down the problem into simpler parts until it's at just the right level for them. Socratic utterances are utterances that guide the user and do not give them the solution directly. In each of your responses, provide a comprehensive list of Socratic responses that you can give to the user to help them solve the problem on their own, based on the conversation so far.

Prompt:

> <problem>
>
> `{{Problem Description}}`
>
> </problem>
>
> <bug_code>
>
> `{{Buggy Code}}`
>
> </bug_code>
>
> <bug_desc>

---

[3] https://openai.com/research/gpt-4

`{{Bug Description}}`

</bug_desc>

<bug_fixes>

`{{Bug Fixes}}`

</bug_fixes>

<unit_tests>

`{{Unit Tests}}`

</unit_tests>

User: {{*User Turn 1*}}

Assistant: {{*Assistant Turn 1*}}

...

User: {{*User Turn N*}}

<code>

{{*Code State at Turn N*}}[a]

</code>

Respond to the user with all possible distinct Socratic utterances that guide the user to discover and fix the bug described between ‘<bug_desc>’ and ‘</bug_desc>’. Student code is written between ‘<code>’ and ‘</code>’ throughout the conversation. Utterances that have the same meaning but different words are considered duplicates. Assume that the student has run the test cases.

1.

---

[a] Included only if turn N has a code state.

APPENDIX B: McMining Appendix

This appendix contains all the prompts, implementation details, and evaluation details for the McMining Chapter 3.

B.1 The McInject Tool

The McInject tool leverages Claude-Sonnet-4.5 with extended thinking enabled to implement the misconception injection task. We use temperature 1.0, max_tokens 6000 (4000 output + 2000 thinking budget), and a 2000 token thinking budget. The tool uses zero-shot prompting with structured XML output format, where the LLM is instructed to modify the correct solution such that it appears as if written by a student who genuinely believes their approach is correct, despite holding the specified misconception.

To enhance generation quality, the prompt includes illustrative examples alongside misconception descriptions. Each misconception in the dataset includes a concrete example demonstrating how the false belief manifests in code, which helps guide McInject to produce more realistic student-like code. The McInject prompt template is shown in Figure B.1.

To further improve code quality, we implement an iterative refinement process. After McInject generates code, an LLM-as-judge evaluates whether the code properly exhibits the target misconception (see Section B.5.1). If the judge provides actionable feedback indicating the misconception is not clearly exhibited, this feedback is appended to the original conversation context, and McInject is prompted to refine the code. This feedback loop can iterates once.

A key feature of McInject is its ability to handle incompatible cases where a misconception cannot be meaningfully applied to a given solution. Misconceptions are considered

Your Task
You will be given as input a programming *problem*, *code* implementing a solution, and the description of a *misconception*. A misconception is a false belief that the student holds about some programming language construct. Note that, misconceptions do not always result in buggy code. Some misconceptions lead to stylistic differences or inefficiencies rather than errors. For example, a student who believes "all variable identifiers must use only one letter" might write working but less readable code.
Modify the code such that it looks as if written by a student who has that misconception. The student genuinely believes that this modified code solves the problem correctly.

Input Format
Problem Description: [problem_description]
Given Implementation:

```
[correct_solution]
```

Misconception Description:

```
[misconception_description]

**Example:**
[misconception_example]
```

Output Format

```
<code>
[The complete modified Python code exhibiting the
misconception]
</code>
```

If the misconception relates to language constructs that are not present in the given solution, output:

```
<code>
NONE
</code>
```

Figure B.1: Prompt template for MCINJECT tool.

incompatible when they relate to language constructs or concepts that are not present or relevant in the solution. For example, a misconception about loop behavior would be incompatible with a solution that contains no loops. In such cases, the tool indicates inapplicability rather than forcing inappropriate modifications.

All code samples produced by MCINJECT undergo post-processing where inline comments are automatically removed to ensure clean output. This process uses Python's tokenize module and includes syntax validation to maintain code correctness.

To validate the quality of generated code samples by McInject, we employed LLM-as-judge evaluation showing that 90.3% of the code samples (1,063 out of 1,177 generated samples) successfully exhibit their target misconceptions. We also employ manual evaluation of 38 samples composed of a misconception and a problem-solution pair that were deemed to be inapplicable by McInject, and observe 100% agreement with human judgment.

We note that this is a conservative estimate of McInject's performance, since the LLM-as-judge discards code samples that do exhibit the misconception, yet are both correct and natural, making the misconception extremely difficult to detect. Manual evaluation of 88 code samples demonstrates 96.6% agreement between LLM judgment and human assessment, confirming the reliability of the LLM-as-judge approach in determining whether a code sample exhibits a misconception description. The LLM-as-judge leveraged Claude-Sonnet-4.5 with the same hyperparameters as McInject.

B.2 The McMiner Tools

This section lists all the prompts that are used by the McMiner tools.

B.3 Misconception Description Writing Guidelines

All misconceptions (composed, adapted, and ooD) were written by hand and include one example of a code snippet exhibiting that misconception. Figure B.4 shows the writing guidelines used to create the misconception bank.

Key Terminology

A **programming misconception** refers to a false belief that a student holds about some programming language construct or built-in function in Python. Programming misconceptions can be about the syntax or the semantics of constructs in the Python programming language. They should not be about concepts in the problem description.

Your Task
Given a problem description and student code that attempts to solve that problem, identify a most likely programming misconception that is exhibited by that code, if any.

Input Format
Problem Description: {problem_description}
Student Code:

```
{student_code}
```

Output Format

```
<misconception>
<description>[Describe misconception, starting
with "The student believes"]</description>
<explanation>[Explain how the code exhibits
             the misconception]</explanation>
</misconception>
```

If no misconceptions are found, output:

```
<misconception>NONE</misconception>
```

Figure B.2: Prompt template for McMiner-S (single-instance mining). The full template includes additional metadata fields and guidelines.

B.4 Experimental Setup

We evaluated McMiner using three state-of-the-art LLMs through their respective APIs:

- **OpenAI**: We used the o3-mini model with two reasoning effort levels. For low effort, we set reasoning_effort to "low" and max_completion_tokens to 7000 (4000 output + 3000 reasoning). For medium effort, we set reasoning_effort to "medium" and max_completion_tokens to 9000 (4000 output + 5000 reasoning). Note that o-series

Key Terminology

A **programming misconception** refers to a false belief that a student holds about some programming language construct or built-in function in Python. Programming misconceptions can be about the syntax or the semantics of constructs in the Python programming language. They should not be about concepts in the problem description. For example, "Student believes range(n) produces values from 1 to n inclusive" is a valid programming misconception about Python's range() function, while "Student thinks natural numbers can be negative" is not a programming misconception. Also, while misconceptions often cause bugs, sometimes they do not. For example, "Student believes all variable names need to have exactly one letter" is a programming misconception that does not necessarily cause a bug.

Your Task

Input: a set of <problem description, code> pairs, where the code in each pair attempts to solve the corresponding problem. Output: the description of a programming misconception that is exhibited by one or more code samples in the input set. While the input set will often contain code samples that show no misconception, try to identify a misconception that is exhibited by most code samples in the input set. If you cannot find any misconception, output NONE.

Input Format

Problem Description: {problem_description}
Student Code:

```
{student_code}
```

Output Format

```
<misconception>
<description>[Describe misconception, starting
with "The student believes"]</description>
<explanation>[Explain how the code exhibits
             the misconception]</explanation>
</misconception>
```

If no misconceptions are found, output:

```
<misconception>NONE</misconception>
```

Figure B.3: Prompt template for McMiner-M (multi-instance mining). The full template includes additional metadata fields and guidelines.

models do not support temperature configuration.

- **Anthropic**: We used the claude-sonnet-4-5 model. For non-reasoning experiments, we used temperature 0.1 and max_tokens 4000. For reasoning-enabled experiments,

Operational definition: A concise, one-sentence statement of the student's incorrect belief about a programming construct, operation, or API, excluding the correct concept.

Guidelines

- **State ONLY the incorrect belief**: Use "Student believes that..." without mentioning or implying the correct construct, concept, or fix.
- **Avoid line numbers**: Reference only the operation or built-in/API involved, not specific line numbers.
- **Be concrete and specific about the construct**: Mention the relevant operation, method, or construct (e.g., `list.pop`, string indexing, `range` bounds, conditional statements).
- **Focus on ONLY one construct**: Write the misconception in terms of only one construct, operation, or API. Avoid general misconceptions applicable to multiple constructs.
- **Generalize the misconception**: The misconception should be applicable to multiple implementations, not specific to a single problem or implementation.
- **Exclude inputs, outputs, and fixes**: Do not include example inputs, expected/actual outputs, or the bug fix.
- **Keep it concise**: One clear sentence that maps directly to what the student implemented.

Figure B.4: Misconception description writing guidelines.

we used temperature 1.0 (as required by Anthropic for extended thinking), max_tokens 6000 (4000 output + 2000 thinking budget), and enabled extended thinking with a 2000 token budget.

- **Google Gemini**: We used the gemini-2.5-flash model with temperature 0.1 and max_tokens 4000. For reasoning-enabled experiments, we increased max_tokens to 6000 (4000 output + 2000 thinking budget) and enabled thinking with a 2000 token budget.

All experiments used individual request processing (no batching) with a zero-shot prompting strategy. For evaluation, we used the benchmark dataset $\mathcal{D}$ described in Section 4, containing the 339 bags of code samples.

To evaluate whether predicted misconceptions semantically match ground truth (GT) misconceptions, we employed an LLM-as-judge approach using Claude Sonnet-4.5 with reasoning capabilities. The evaluation uses identical hyperparameters as MCINJECT. Claude evaluates whether the predicted misconception description matches the intended misconception.

To credit novel misconception discoveries not present in our benchmark, we implement a novelty-aware evaluation approach. For predictions that do not match GT, we use the same LLM-as-judge validation from MCINJECT to determine whether the code samples accurately exhibit the predicted misconception. If validated, these are counted as novel true positives.

B.5 Evaluation Prompts

This section discusses all the prompts used for evaluation in this paper.

B.5.1 McInject LLM-as-judge Prompt

To validate that code samples generated by MCINJECT properly exhibit their intended misconceptions, we employ an LLM-as-judge evaluation. The judge is given a misconception description with an example, and a code sample to analyze. The task is to determine whether the code exhibits the misconception, providing a binary judgment (Y/N) and optional feedback.

When used in the iterative refinement process, the feedback mechanism works as follows: if the judge determines the misconception is not clearly exhibited and provides specific feedback (i.e., feedback is not “NONE”), this feedback is used to create a multi-turn conversation. The original MCINJECT prompt and its initial response are preserved, and the feedback is appended as a new user message instructing MCINJECT to improve the code based on the critique. This creates a conversational refinement loop where MCINJECT can iteratively improve its output. The process terminates when either the judge confirms the misconception is exhibited (feedback is “NONE”), or the iteration limit is reached.

Importantly, the prompt emphasizes that misconceptions do not necessarily cause bugs—code can be syntactically and logically correct while still exhibiting a false belief pattern. Figure B.5 shows the prompt template used for this evaluation.

Input Format
The Misconception

```
<misconception>
Description: {misconception_description}
Example: {misconception_example}
</misconception>
```

The Code to Analyze

```
<code>
{code_to_analyze}
</code>
```

Your Task
Determine whether this code exhibits the misconception described above.

Key Understanding
A misconception does NOT necessarily induce bugs or errors! Code can be:

- **Syntactically correct** (no syntax errors)
- **Logically correct** (produces expected output)
- **Yet still exhibit a misconception** (shows the student holds a false belief)

Analysis Guidelines

1. **Understand the misconception deeply**: What incorrect belief does the student have? What coding patterns would reveal this belief?
2. **Analyze the code systematically**: Look for patterns that match the misconception. Check if the code structure reflects the incorrect belief.
3. **Focus on the belief, not the outcome**: Does the code structure suggest the student holds this false belief? Even if the code works, does it show the misconception pattern?

Output Format

```
<answer>
<exhibits_misconception>Y or N</exhibits_misconception>
<feedback>[Optional feedback]</feedback>
</answer>
```

Figure B.5: Prompt template for LLM-as-judge evaluation of MCINJECT generated code samples. The full prompt template includes metadata fields to output such as rationale and confidence level.

B.5.2 McMiner Semantic Matching Prompt

To evaluate McMiner predictions, we use an LLM-as-judge to determine whether a predicted misconception semantically matches the ground truth misconception. The judge is provided with the ground truth misconception, the predicted misconception, and the code samples that were analyzed. The task is to assess whether both descriptions capture the same fundamental programming misunderstanding, accounting for natural variations in wording. The evaluation considers core concept alignment, evidence in the code samples, and semantic equivalence. Figure B.6 shows the prompt template used for this evaluation.

Input Format
Ground Truth Misconception

```
{ground_truth}
```

Predicted Misconception

```
{predicted_misconception}
```

Code Samples Analyzed

```
{code_samples}
```

Task
Determine if the predicted misconception accurately captures the same conceptual misunderstanding as the ground truth. Consider:

1. **Core Concept Match**: Are they describing the same fundamental misunderstanding about programming concepts?
2. **Evidence Alignment**: Does the predicted description align with what's shown in the code samples?
3. **Semantic Equivalence**: Minor wording differences are acceptable if the core concept matches.
4. **Example Reference**: If provided, use the misconception example to better understand the typical manifestation of this misconception.

Special Cases

- If ground truth is "NO MISCONCEPTION" and prediction found no misconceptions, this is a match.
- If ground truth is "NO MISCONCEPTION" but prediction found misconceptions, this is NOT a match.
- If ground truth describes a misconception but prediction found none, this is NOT a match.

Output Format

```
<evaluation>
<match>true or false</match>
</evaluation>
```

Figure B.6: Prompt template for semantic matching evaluation of McMiner predictions. The full prompt template includes metadata fields to output such as confidence level and explanation.

APPENDIX C: Reasoning Trajectory Appendix

This appendix contains all the prompts, implementation details, evaluation details, detailed qualitative analysis, LLM hyper-parameters, and other supplementary materials for the Reasoning Trajectory Chapter (§4).

C.1 Problem and Misconception Dataset Sources

We use a problem-solution set containing 558 solutions across 501 problems from the McMining dataset [87]. This dataset contains 438 problems from MBPP [150], 27 from Socratic Debugging [69, 70], 8 from Auckland [151], 6 from FalconCode [152], and 19 handwritten solutions. The McMining dataset also contains 67 misconceptions, with 40 of them being bug-inducing misconceptions. We select all 40 bug-inducing misconceptions from the dataset and use them for our experiments.

C.2 Pairing Misconceptions with Program Solutions

To ensure plausibility in LLM-generated corrupted code, we developed an automated construct-based pairing algorithm that generates 250 high-quality (misconception, solution) pairs. These pairs serve as input to McInject, ensuring each misconception is matched with solutions containing the necessary programming constructs. Without proper construct matching, two failure modes occur: (1) *inapplicable misconceptions*, where the pattern cannot be applied (e.g., a class-related misconception on code without classes), and (2) *implausible code*, where the LLM seems to exhibit the misconceptions but lacks logical coherence. A student with that misconception would not have written such code and genuinely believed that it would solve the programming problem.

Phase 1: Programming Construct Extraction We extract fine-grained programming constructs from each solution using a combination of abstract syntax tree (AST) parsing and pattern matching. Our implementation uses Python's `ast` module to parse the syntax tree and identify control flow structures (loops, conditionals), function definitions, operators, and data structures. Additionally, we employ regular expressions (`re` module) to detect patterns not easily captured by AST analysis, such as method chaining (e.g., `x.a().b()`), specific string methods (e.g., `.upper()`, `.split()`), and particular operator combinations relevant to precedence misconceptions.

This process analyzes all 558 solutions across 501 problems from our problem-solution set described in Appendix C.1. The extraction produces solution-level constructs, identifying over 80 distinct construct types, including specific language features like `list.append`, `str.split`, recursion patterns, and class initialization.

Phase 2: Construct-Overlap Pairing Given extracted constructs, we generate 250 (misconception, solution) pairs through semantic alignment based on construct overlap. For each misconception m hand-annotated with *related_constructs* (e.g., "for loops", "indexing", "range function"), we compute an overlap score with each solution s as:

$$\text{score}(m, s) = |\text{constructs}[m] \cap \text{constructs}[s]| \tag{C.1}$$

For regular misconceptions, only pairs with score$(m, s) \geq 1$ are considered, ensuring compatibility. However, 16 misconceptions require *special case handling* where code pattern matching can override the overlap requirement: for instance, recursion misconceptions require actual recursive function calls verified through code analysis, class misconceptions require `__init__` methods, and operator precedence misconceptions require specific operator combinations like + and / in the same expression. For these special cases, solutions are accepted if they either have construct overlap *or* satisfy the required code pattern.

Algorithm 3 Construct-Based Misconception-Solution Pairing

```
Require: Solutions $S$ with extracted constructs, Misconceptions $M$ with constructs, Target count $N$
Ensure: Pairings $\mathcal{D} = \{(m_i, s_i)\}_{i=1}^{N}$
Phase 1: Extract Constructs
for each solution $s \in S$ do
    constructs$[s] \leftarrow$ ExtractConstructs$(s)$                ▷ AST + regex
Phase 2: Generate Pairs
used $\leftarrow \emptyset$                                          ▷ Track used solutions
$\mathcal{D} \leftarrow \emptyset$                                   ▷ Result pairings
$i \leftarrow 0$                                                     ▷ Round-robin index
while $|\mathcal{D}| < N$ do
    $m \leftarrow M[i \bmod |M|]$                                    ▷ Current misconception
    candidates $\leftarrow \emptyset$
    for each solution $s \in S \setminus$ used do
        score $\leftarrow |$constructs$[m] \cap$ constructs$[s]|$
        if score $\geq 1$ or IsSpecialCase$(m, s)$ then
            candidates $\leftarrow$ candidates $\cup \{(s, \text{score})\}$
    if candidates $\neq \emptyset$ then
        $s^* \leftarrow \arg\max_{(s,\text{score}) \in \text{candidates}}$ score
        $\mathcal{D} \leftarrow \mathcal{D} \cup \{(m, s^*)\}$
        used $\leftarrow$ used $\cup \{s^*\}$
    $i \leftarrow i + 1$
  return $\mathcal{D}$
```

The algorithm employs round-robin allocation, cycling through misconceptions while selecting the highest-scoring unused solution for each. This ensures diversity (each solution used at most once) while handling varying construct availability. Common constructs like loops yield 13-14 pairs per misconception, while rare constructs like method chaining yield only 2-4 pairs out of the 250 pairs.

Algorithm 3 presents the complete procedure. The algorithm uses an auxiliary function

IsSpecialCase(m, s) that returns true if misconception m and solution s match one of 16 special case patterns requiring code verification (e.g., recursion misconceptions paired with solutions containing actual recursive function calls, class misconceptions with solutions containing `__init__` methods, or operator precedence misconceptions with specific operator combinations like + and `/`). The output is 250 validated pairs stored with metadata, including overlap scores and matching constructs for reproducibility and analysis. Our implementation uses standard Python libraries: `ast` for abstract syntax tree parsing, `re` for regular expression pattern matching, and `json` for data serialization. The entire process is fully deterministic.

C.3 LLM Hyper-parameters

Our experiments evaluate three state-of-the-art LLMs via their respective APIs, each configured with model-specific parameters:

- **OpenAI GPT-5**: We employ the `gpt-5` and `gpt-5-mini` models using the Responses API, which provides built-in reasoning capabilities. The model supports four reasoning effort levels: *minimal*, *low*, *medium*. We avoid using the *high* reasoning effort level since we observed issues with the batching API. We set max_output_tokens to 4000 and configure text verbosity to *medium*. Unlike traditional models, the GPT-5 family does not support temperature configuration, as reasoning effort directly controls the model's deliberation intensity.
- **Anthropic Claude**: We utilize the `claude-sonnet-4-5` and `claude-haiku-4-5` models with two operational modes. For standard generation, we apply a temperature of 0.1 and max_tokens 4000. When extended thinking is enabled, we increase temper-

ature to 1.0 (as mandated by Anthropic's API), allocate an additional 2000 tokens for the thinking budget (yielding 6000 total tokens), and activate the thinking mode with budget_tokens set to 2000.

- **Google Gemini**: We use the `gemini-2.5-flash` and `gemini-2.5-pro` models with temperature 0.1 and max_output_tokens 4000 for baseline experiments. For reasoning-enabled experiments, we augment max_output_tokens by an additional 2000 tokens (totaling 6000), and configure thinking_config with include_thoughts enabled and thinking_budget set to 2000.

All experiments utilize the LLM provider APIs. The complete evaluation prompt specifications are provided in Appendix C.5 while the reasoning trajectory generation prompt template is provided in §4.4.1 and the Socratic turn generation is provided in §4.4.2.

C.4 Failed Test Case Generation

For each buggy code sample in our dataset, we generate a concise description of the simplest failing test case to serve as input for reasoning trajectory generation. To do so, we execute all the unit tests associated with each problem using a Python interpreter and capture detailed execution results including passed tests, failed tests with output mismatches, runtime errors, and syntax errors.

Figure C.1 shows the prompt template for failed test case description generation. Given the problem description, buggy code, and execution results, the LLM selects the simplest failing test according to a priority order: syntax errors first (affecting all tests), then runtime errors, then logical errors with simplest inputs. The output is a one-sentence description with description writing conventions based on the error type, ensuring consistency across the dataset.

Your Task
Given a Python problem, buggy code, and execution results, select the simplest failing test case and write a concise description of how it fails.

Selection Strategy

1. **Syntax errors**: If present, all tests fail the same way
2. **Runtime errors**: Select the test with simplest inputs that raises the error
3. **Logical errors**: Choose test with most basic arguments (single values, small numbers, edge cases)
4. **First failing test**: If multiple tests fail similarly, choose the first one

Output Format Conventions

- **Logical errors**: "When called as [function_call], the function returns [actual]; whereas the expected result is [expected]."
- **Runtime errors**: "When called as [function_call], the function raises [ErrorType] on line [N]."
- **Syntax errors**: "When called as [function_call], the function produces a SyntaxError on line [N]."

Input Format

```
Problem Description: [problem_description]
Buggy Code: [buggy_code]
Execution Results: [execution_results]
```

Figure C.1: Prompt template for failed test case description generation. The full template includes detailed execution result formats and worked examples for each error type.

This automated approach ensures that each buggy code sample is paired with a clear, consistent description of its simplest failure, providing a focused starting point for reasoning trajectory generation. We task the LLM to select the simplest failing test and describe it in a concise manner. This reduces unnecessary complexity while debugging and code tracing in the reasoning trajectory. This ensures that the Socratic conversation focuses on the core misconception and avoids unnecessary complexity. The LLM used to generate the failed test case description is `Claude Sonnet-4.5` with temperature 0.1, disabled reasoning, and max_tokens 4000.

C.5 Language Model Evaluation Prompts

To ensure the quality of generated reasoning trajectories and Socratic conversations, we employ an LLM-as-judge evaluation approach. We use `Claude Sonnet-4.5` with extended thinking enabled as the evaluator model, configured with temperature 1.0 and max_tokens 8000. The evaluation uses prompting with structured criteria that enable systematic assessment of both logical rigor and pedagogical appropriateness.

This automated evaluation approach enables scalable assessment across larger datasets, maintains consistency in applying complex evaluation criteria, and produces structured feedback that can guide LLMs through iterative refinement. The evaluation prompts are carefully designed to operationalize abstract quality requirements into concrete and verifiable criteria.

C.5.1 Reasoning Trajectory Evaluation

To validate that generated reasoning trajectories serve as rigorous logical proofs by counterexample, we evaluate them across three hierarchical categories. Figure C.2 shows a simplified evaluation prompt template since the full template is too long to fit in the page. The evaluation prompt is a zero-shot prompting approach which does not include any worked examples.

The evaluation framework assesses **Logical Soundness** through five criteria: valid starting point from the failed test, deductively valid inferences without abduction or logical leaps, sound contradiction of the target misconception, complete causal chain from observation to

Your Task
Evaluate whether a reasoning trajectory (RT) serves as a rigorous, logical proof by counterexample that contradicts a student misconception. An RT is **VALID** only if it passes all criteria in all three categories below.

Category 1: Logical Soundness

- **Valid Starting Point**: Begins with verifiable fact from failed test
- **Deductively Valid**: Each step follows necessarily from prior steps and Python semantics. No abduction or logical leaps.
  Does not assume programming knowledge that directly contradicts the misconception.
- **Sound Contradiction**: Establishes facts incompatible with misconception.
- **Complete Causal Chain**: Unbroken chain from observation to contradiction
- **Execution Tracing**: Traces program execution to deduce concrete facts

Category 2: Step Construction & Precision

- **Clear Boundaries**: Each step is a distinct logical unit
- **Precision**: Uses specific line numbers, variable names, values
- **Proper Citation**: Non-adjacent dependencies explicitly cited
- **Technical Accuracy**: All claims about Python constructs are correct

Category 3: Formatting & Focus

- **Sequential Labeling**: All steps labeled sequentially (A.1, A.2, ...)
- **Focus on Misconception**: Exclusively focused on disproving target misconception

Output Format

```
{
  "valid": true/false,
  "categories": {
    "logical_soundness": true/false,
    "step_construction_and_precision": true/false,
    "formatting_and_focus": true/false
  },
  "comments": "[Evaluation rationale]",
  "feedback": "[Actionable suggestions or NONE]"
}
```

Figure C.2: Prompt template for LLM-as-judge evaluation of reasoning trajectories. An RT is valid only if all three categories pass. The full template includes detailed criterion descriptions and scoring instructions.

contradiction, and proper execution tracing to deduce concrete facts. Although reasoning steps may use general knowledge about Python, mathematics, or other domains to deduce facts about program execution, they must not assume programming knowledge that directly contradicts the misconception. For example, if the student believes `range(n)` starts at 1, the reasoning cannot assume it starts at 0, since the student holding the misconception would disagree with that assumption in a Socratic conversation. The **Step Construction & Precision** category evaluates clear step boundaries, precision in referencing code elements, proper citation of dependencies, and technical accuracy of all claims. Finally, **Formatting & Focus** ensures sequential step labeling and exclusive focus on contradicting the target misconception.

A reasoning trajectory is considered valid only if all criteria in all three categories pass. This hierarchical scoring mechanism ensures that RTs meet both the logical rigor required for sound proofs and the pedagogical clarity needed for effective student guidance. The judge outputs structured JSON with binary scores for each category, detailed comments explaining the evaluation rationale, and actionable feedback for invalid RTs.

C.5.2 Socratic Turn Evaluation

To ensure that generated Socratic conversations effectively guide students through the reasoning trajectory, we evaluate each teacher utterance using a two-criterion framework. Figure C.3 shows a simplified evaluation prompt template. The full template includes 5 fully worked examples.

The first criterion, **Prompts the Correct Inference**, ensures that the teacher's question guides the student to articulate the specific inference from the target RT step. The student's

Your Task

Evaluate whether a Teacher utterance in a Socratic conversation effectively guides a student to articulate the inference from a specific RT step. A teacher utterance is **VALID** only if it satisfies both criteria below.

Criterion 1: Prompts the Correct Inference

The teacher's question must guide the student to articulate the key inference from the specific RT step it claims to prompt. The student's response should contain the statement proven in that step, and only that step. Questions may state facts from previous steps but must prompt the new inference at the target step.

Criterion 2: Does Not State the Inference Directly

The teacher must ask a question requiring reasoning. The teacher should not provide the answer or state the conclusion. Questions can be general ("What's the issue?") or specific, as long as they require the student to think and derive the answer rather than merely confirm a stated fact.

Evaluation Process

1. Read RT step A.X to understand the target inference
2. Read RT steps A.1 through A.X-1 for established facts
3. Read the teacher utterance and student response
4. Evaluate against both criteria
5. Valid only if both criteria pass

Output Format

```
{
  "valid": true/false,
  "criteria_scores": {
    "prompts_correct_inference": true/false,
    "does_not_state_inference": true/false
  },
  "comments": "[Evaluation explanation]",
  "feedback": "[Suggestions or NONE]"
}
```

Figure C.3: Prompt template for LLM-as-judge evaluation of Socratic teacher utterances. A teacher utterance is valid only if it prompts the correct RT step without stating the inference directly. The full template includes worked examples demonstrating both valid and invalid utterances.

response should contain the statement proven in that step, and only that step. Questions may reference facts established in previous steps but must prompt the new inference at the target step. The second criterion, **Does Not State the Inference Directly**, ensures the teacher asks a genuine question requiring reasoning rather than stating the conclusion and requesting confirmation.

A teacher utterance is considered valid only if it satisfies both criteria. The evaluation process follows a systematic procedure: read the target RT step to understand the required inference, review previous steps for context, examine the teacher utterance and student response, and evaluate against both criteria. The judge outputs structured JSON with binary scores for each criterion, explanatory comments, and actionable feedback for invalid utterances.

C.6 Reasoning Trajectory Generation Qualitative Analysis

To complement our quantitative evaluation, we conduct a qualitative analysis of reasoning trajectory generation. Using the RT evaluation criteria established in Appendix C.5.1, we perform manual evaluation on 30 reasoning trajectories created from 10 samples from each of three models. This section presents both successful and unsuccessful patterns observed.

C.6.1 Success Patterns

We identified three key patterns that characterize successful reasoning trajectory generation from the manually evaluated samples, demonstrating rigorous logical proof by counterexample.

► **Exhausting Alternative Possibilities.** Effective reasoning trajectories enumerate all possible scenarios and systematically eliminate scenarios that contradict observed behavior.

Consider an example where a student has written:

```
def calculate_average(x, y):
    return x + y / 2
```

The student incorrectly believes that + has higher precedence than `/`. When called as `calculate_average(1, 3)`, the function returns 2.5 instead of the expected 2.0. The reasoning trajectory establishes:

Step A.1: The failed test states that `calculate_average(1, 3)` returns 2.5. So with `x = 1` and `y = 3`, the expression on line 2, `x + y / 2`, evaluates to 2.5.

Step A.2: There are no parentheses in line 2, so the only two possible groupings of `x + y / 2` are: (1) `(x + y) / 2`, or (2) `x + (y / 2)`.

Step A.3: Compute `(x + y) / 2` with `x = 1` and `y = 3`: `(1 + 3) / 2 = 4 / 2 = 2.0`.

Step A.4: If + had higher precedence than `/`, then line 2 would be evaluated as `(x + y) / 2`, which we computed to be 2.0 (A.3). But the actual result is 2.5 (A.1). Therefore, + is not evaluated before `/` in this expression.

This approach demonstrates systematic elimination: the RT enumerates all possible interpretations (Step A.2), computes the result under each interpretation (Step A.3), and rules out the interpretation matching the misconception by contrasting it with observed behavior (Step A.4). The exhaustive enumeration ensures no logical gaps remain.

► **Concrete Execution Tracing.** Successful reasoning trajectories ground abstract reasoning in concrete test values, tracing program execution with specific inputs throughout the logical chain. In the same example above, the RT uses the specific values `x = 1` and `y`

`= 3` from the failed test consistently across all steps. Step A.1 establishes these concrete values from the failed test. Step A.3 computes the concrete result `(1 + 3) / 2 = 4 / 2 = 2.0` for the first grouping. Step A.5 (not shown above) verifies the alternative grouping `1 + (3/2) = 2.5` matches the observed output. This concrete tracing ensures every deductive step is verifiable against observable program behavior rather than relying on abstract reasoning about Python semantics. The specificity eliminates ambiguity and makes each logical inference checkable.

► **Clear Contradiction.** Effective reasoning trajectories explicitly structure the contradiction between the misconception and observed behavior. Step A.4 in the example above demonstrates this structure: it first states the implication of the misconception ("If + had higher precedence than `/`, then line 2 would be evaluated as `(x + y) / 2`"), then computes the result under that assumption ("which we computed to be 2.0"), and finally contrasts this with actual observed behavior ("But the actual result is 2.5"). The explicit "If...then...But" structure makes the logical contradiction transparent. This pattern ensures the reasoning trajectory achieves its primary purpose: proving the misconception leads to predictions incompatible with observed program behavior.

C.6.2 Failure Patterns

Several failure modes emerged in manual evaluation, each revealing different challenges in constructing logically sound reasoning trajectories.

► **Using Knowledge that Contradicts the Misconception.** The most pedagogically damaging failure pattern occurs when reasoning trajectories rely on programming knowledge that directly contradicts the target misconception. Consider an example where a student has written:

```python
def top_k(lst, k):
    result = []
    for i in range(k):
        result.append(max(lst))
        lst.pop(max(lst))   # Line 5
    return result
```

The student incorrectly believes that the `.pop()` method takes a value to be deleted from the list. When called as `top_k([1, 2, 3, 4, 5], 1)`, the function raises an `IndexError` on line 5. The reasoning trajectory contains:

Step A.5: In Python, `list.pop(i)` removes and returns the item at index `i`; if `i` is outside the valid index range for the list, it raises `IndexError`. Therefore, the observed `IndexError` on `lst.pop(5)` means the argument 5 was used as an index, not as a value.

This step explicitly states how `list.pop()` actually works, directly contradicting the student's belief. A student holding the misconception would reject this premise in a Socratic conversation. The RT evaluation criterion states that reasoning steps must not assume programming knowledge that directly contradicts the misconception. The correct approach would prove that interpreting the argument as a value leads to a contradiction, without stating how `pop()` actually works.

► **Abductive Reasoning and Logical Leaps.** Reasoning trajectories sometimes employ abductive reasoning (inference to the best plausible explanation) rather than deductive proof, leaving logical gaps. Consider an example where a student has written:

```
def count_words(sentence):
    words = 0
    space_mode = True
    for i in range(1, len(sentence)):  # Line 4
        if sentence[i] == ' ':
            if not space_mode:
                words += 1
            space_mode = True
        else:
            space_mode = False
    if not space_mode:
        words += 1
    return words
```

The student incorrectly believes string indexing starts at 1. When called as `count_words("I love Python")`, the function returns 2 instead of 3. After establishing that the loop executes from `i = 1` to `i = 12` and that `words` is incremented exactly once during the loop, the reasoning trajectory contains:

Step A.9: At the start of the loop when `i = 1`, `space_mode` is `True` (initialized on line 3). If `sentence[1]` is a space, line 6's condition `not space_mode` would be `False`, so line 7 would not execute. For the algorithm to eventually count only 2 words while “I love Python" has 3 words, and for line 7 to execute exactly once, `sentence[1]` must be a space.

This reasoning works backward from the observed output to infer that `sentence[1]` must be a space. However, it does not prove this is the only possibility that produces `words = 2`, leaving the logical chain incomplete. The RT must explain: (1) what happens if `sentence[1]` is NOT a space (e.g., if it’s ’I’)? and (2) why would that scenario fail to produce the observed output of ‘words‘ = 2?

Another example demonstrates this pattern more concisely. In the `top_k` example above, a different reasoning trajectory contains:

Step A.7: Since calling `lst.pop(5)` raises an `IndexError`, and the number 5 is an invalid index for `lst`, the `.pop()` method must be interpreting the argument 5 as an index, not as a value.

The phrase “must be interpreting" reveals abduction. The step infers the most likely explanation but does not prove it deductively. It is theoretically possible for `pop()` to use the argument in a different way while still raising `IndexError`. A valid approach would prove that if 5 were interpreted as a value to remove, no error would occur (since 5 exists in the list), establishing contradiction.

► **Technical Inaccuracy.** Reasoning trajectories sometimes contain technically incorrect claims about programming constructs, undermining their logical validity. Consider an example where a student has written:

```python
def is_palindrome(string):
    rev_string = ''
    for i in string:
        rev_string = i + rev_string
    if rev_string = string:  # Line 5
        return True
```

```
    else:
        return False
```

The student incorrectly believes the = operator is used for equality comparison. When called as `is_palindrome("racecar")`, the function produces a `SyntaxError` on line 5. The reasoning trajectory contains:

Step A.3: In Python, an `if` statement requires a conditional expression following the `if` keyword. A conditional expression is something that can be evaluated to determine if it is true or false.

This statement uses the term “conditional expression" incorrectly. According to Python documentation[1], a conditional expression refers specifically to the ternary operator (`x if C else y`), which is an expression that evaluates to a value. The RT appears to mean “boolean expression" or “condition," but the misuse of technical terminology contradicts authoritative documentation. Although the RT later defines the term differently (“something that can be evaluated to determine if it is true or false"), this redefinition itself violates technical accuracy. Such inaccuracies undermine the RT’s credibility and may confuse students learning precise programming terminology.

► **Incomplete Logical Chains.** Reasoning trajectories sometimes skip necessary intermediate steps, leaving gaps in the causal chain from observation to contradiction. In the same `is_palindrome` example above, the reasoning trajectory establishes:

Step A.4: The statement on line 5 uses the single equals sign (=). In Python, the = operator is the assignment operator. Its function is to assign the value on its right to the

---

[1] https://docs.python.org/3/reference/expressions.html#conditional-expressions

variable on its left.

Step A.5: Because the = operator is for assignment and not for evaluation, the expression `rev_string = string` is an assignment statement, not a conditional expression.

The logical leap occurs between these steps. Step A.4 establishes that = is used for assignment. Step A.5 concludes that = cannot be used for comparison. However, the RT does not prove that an operator cannot serve both purposes. A student might reasonably ask: “Why can’t = be used for both assignment and comparison in different contexts?" The missing step must establish that Python operators are unambiguous and cannot serve dual purposes in the same statement context. Without this intermediate link, the causal chain from “= is for assignment" to “= is not for comparison" remains incomplete.

C.7 Socratic Utterance Generation Qualitative Analysis

To complement our quantitative evaluation, we conduct qualitative analysis of Socratic utterance generation. We manually evaluated 88 teacher utterances evaluated by both an LLM judge and a human expert created from 15 reasoning trajectories, 5 from each of three models: Claude Sonnet-4.5 with reasoning, Gemini-2.5-pro with reasoning, and GPT-5 with medium reasoning effort. This section presents both successful and unsuccessful patterns observed, while evaluating against the Socratic turn evaluation criteria established in Appendix C.5.2.

C.7.1 Success Patterns

We identified four key patterns that characterize successful Socratic conversation generation across the manually evaluated samples.

► **Accurate RT Step Alignment.** Across all manually evaluated teacher utterances, we did not observe any cases where a teacher question prompted content from a completely different RT step than requested. For instance, we found no instances where a question marked for prompting Step A.5 actually prompted for Step A.7's conclusion. This indicates alignment between generated questions and their intended reasoning steps, demonstrating that LLMs reliably understand the step-by-step structure of reasoning trajectories and can write Socratic utterances specific to a logical inference.

► **Step Reference Integration.** Effective Socratic questions occasionally need to explicitly incorporate facts established in prior RT steps while prompting a new inference. LLMs successfully synthesize these references into a single, coherent question. Consider the following example where the student has written buggy code that incorrectly assumes `str.replace()` modifies strings in place:

```
def toxNGLXSH(sen):
    vowels = ["a"]
    for i in vowels:
        if i.islower():
            sen.replace(i, "x")  # Line 5
        else:
            sen.replace(i, "X")
    return sen
```

The reasoning trajectory establishes:

- **Step A.2:** The loop executes exactly once with `i = 'a'`
- **Step A.3:** Since `i = 'a'` (A.2) and `'a'.islower()` evaluates to `True`, the condi-

tion on line 4 is satisfied, so line 5 is executed

The teacher question targeting Step A.3 is: “Given that `i = ‘a’`, which branch of the if-else statement executes?”

This question effectively integrates the fact from Step A.2 (`i = ‘a’`) as a given, then prompts the student to determine which branch executes. The question requires the student to evaluate the condition `i.islower()` and conclude that line 5 executes, which is precisely the inference in Step A.3. The prior step’s conclusion becomes established context for the new question.

► **Implicit Prompting.** Socratic questions prime students to provide information beyond what is explicitly requested, demonstrating understanding of what constitutes a complete logical step. Consider another example with the same buggy code. The reasoning trajectory step is:

Step A.4: On line 5, the expression `sen.replace(i, “x")` is evaluated. Given that `sen = ‘a’` and `i = ‘a’` (A.2), this evaluates to `‘a’.replace(‘a’, ‘x’).`

The teacher question is simply: “What expression is evaluated on line 5?”

The student responds: “On line 5, the expression `sen.replace(i, “x")` is evaluated. Given that `sen = ‘a’` and `i = ‘a’`, this evaluates to `‘a’.replace(‘a’, ‘x’)`.”

Although the question asks only “what expression is evaluated,” it implicitly expects the student to provide both the abstract expression (`sen.replace(i, “x")`) and its concrete evaluation with values substituted (`‘a’.replace(‘a’, ‘x’)`). The student correctly provides this complete inference without requiring explicit prompting for each component.

► **Generic Opening Questions.** All of the conversations in our manual evaluation begin with broad, open-ended questions such as "What issue are you encountering?" or "What seems to be the problem?" These effectively elicit students' initial observations about failed tests without leading them. Such opening moves are straightforward and natural components of Socratic dialogue, and all models generate them successfully.

C.7.2 Failure Pattern

We identify a primary failure mode in generated Socratic utterances, representing a failure in asking a Socratic question with good pedagogical quality.

► **Stating the Conclusion Directly.** Teacher utterances sometimes reveal the inference students should derive, reducing the question to mere confirmation rather than genuine reasoning. Consider an example where the student has written:

```python
def calculate_average(x, y):
    return x + y / 2
```

The student incorrectly believes that + has higher precedence than `/`. The reasoning trajectory establishes:

Step A.1: When called as `calculate_average(1, 3)`, the function returns 2.5, meaning `x = 1`, `y = 3`, and `x + y / 2` evaluates to 2.5.

. . .

Step A.4: Let's assume the misconception is true: + has higher precedence than `/`.

Step A.5: Applying this assumption, `1 + 3` would be evaluated first, resulting in `4`.

Step A.6: After evaluating the addition, the expression would simplify to `4 / 2`, which equals `2.0`.

Step A.7: Therefore, the assumption that + has higher precedence than / leads to the conclusion that the expression `1 + 3 / 2` evaluates to `2.0` (Step A.6).

The generated teacher question for Step A.7 is: “So, your assumption that addition comes first leads to a final result of `2.0`. How does that compare to what the program actually calculated?”

The first sentence explicitly states the conclusion from Step A.7 before asking the question. The student only needs to compare the values, not derive that the assumption leads to `2.0`. A correct alternative would omit the statement of the conclusion and ask only: “How does that compare to what the program actually calculated?”

C.8 LLM-as-Judge Qualitative Analysis

To complement the quantitative agreement rates reported in the main paper, we conduct qualitative analysis of LLM-as-judge evaluation patterns. We analyze disagreement cases between the LLM judge and human expert across both reasoning trajectory validation and Socratic turn validation to identify failure modes and better understand the judge’s capabilities and limitations.

C.8.1 LLM-as-Judge Failures for Reasoning Trajectory Evaluation

RT validation achieved 76.66% agreement between the LLM judge and human expert, notably lower than Socratic turn validation (96.59%), suggesting more nuanced challenges in evaluating the logical rigor of reasoning trajectories. We analyze disagreement cases to identify patterns where the judge fails.

► **Fails to Detect Knowledge Contradicting Misconceptions.** The judge incorrectly marked reasoning trajectories as valid when they explicitly assume programming knowledge that contradicts the target misconception. In the `top_k` example from the failure patterns analysis, the reasoning trajectory contains:

Step A.5: In Python, `list.pop(i)` removes and returns the item at index `i`; if `i` is outside the valid index range for the list, it raises `IndexError`. Therefore, the observed `IndexError` on `lst.pop(5)` means the argument 5 was used as an index, not as a value.

The human expert marked this RT as invalid because Step A.5 explicitly states how `list.pop()` works, directly contradicting the misconception that `pop()` takes a value to delete. However, the judge marked the RT as valid. The judge appears to miss that this assumption violates the RT evaluation criterion: “Does not assume programming knowledge that directly contradicts the misconception." This represents a critical oversight since this failure pattern is the most pedagogically damaging, rendering the RT unusable in Socratic dialogue where students holding the misconception would reject the contradicting premise.

Another example, in the `count_words` example from the failure patterns analysis, the reasoning trajectory contains:

Step A.9: At the start of the loop when `i = 1`, `space_mode` is `True` (initialized on line 3). If `sentence[1]` is a space, line 6’s condition `not space_mode` would be `False`, so line 7 would not execute. For the algorithm to eventually count only 2 words while “I love Python" has 3 words, and for line 7 to execute exactly once, `sentence[1]` must be a space.

The human expert and the LLM marked this RT as invalid because Step A.9 uses abduction rather than proving deductively that `sentence[1]` is a space. However, the judge’s feedback suggested fixing the RT by “directly observing that in the input string ’I love Python’, the character at index 1 is verifiably a space." This suggestion itself violates RT evaluation rules by assuming that indexing starts at 0 (since it claims index 1 contains a space, which is the second character). The judge was suggesting assuming knowledge that contradicts the misconception to address the abduction issue of this step.

► **Fails to Catch Technical Inaccuracies.** The judge does not verify technical claims about programming constructs against authoritative documentation. In the `is_palindrome` example from the failure patterns analysis, the reasoning trajectory contains:

Step A.3: In Python, an `if` statement requires a conditional expression following the `if` keyword. A conditional expression is something that can be evaluated to determine if it is true or false.

The human expert marked this RT as invalid because “conditional expression" has a specific technical meaning in Python (referring to the ternary operator `x if C else y`) that contradicts how the RT uses the term. The judge marked the RT as valid, missing this technical inaccuracy. The judge appears to lack capability to fact-check domain-specific terminology against Python documentation, accepting the RT’s redefinition of the term as sufficient. This suggests the judge evaluates logical structure without verifying technical accuracy of claims about programming language semantics.

► **Over-Accepts Incomplete Logical Chains.** The judge sometimes accepts reasoning trajectories with missing intermediate steps that leave gaps in the causal chain. In the same `is_palindrome` example, the reasoning trajectory establishes that = is used for assignment (Step A.4) and then concludes that = cannot be used for comparison (Step A.5), without proving that Python operators cannot serve dual purposes. The human expert identified this as an incomplete logical chain, but the judge marked the RT as valid. The judge appears too lenient on what constitutes a “complete causal chain," accepting direct leaps from A to C without requiring explicit proof of the intermediate link B. This suggests the judge may assess whether conclusions follow plausibly rather than whether they follow necessarily from stated premises.

Despite these systematic failures, the judge correctly identifies valid reasoning trajectories that employ sound deductive reasoning, proper execution tracing, and clear contradiction structure. The challenge lies in detecting subtle logical flaws (abduction, missing steps, contradictory assumptions, technical errors) rather than recognizing rigorous proofs. The overall agreement rate with a human expert is high, scoring 76.66%, highlighting the reliability of the LLM-as-judge for this task.

C.8.2 LLM-as-Judge Failures for Socratic Turn Evaluation

We conducted manual evaluation of 88 teacher utterances, comparing the LLM judge (Claude Sonnet-4.5 with extended thinking) against human expert judgments. The overall agreement rate was 96.59%. Error analysis on the disagreement samples reveals some patterns.

C.8.2.1 False Negative: Judge Too Strict

In one case, the judge incorrectly marked a valid teacher utterance as invalid, revealing a systematic issue with how the judge handles conversational framing.

► **Conversational Framing Penalized.** The sample involves a student who has written:

```
class Node:
    def __init__(self, data=None, next=None):
```

```
        self.data = data
        self.next = next

def insert_after(head, prev_data, new_data):
    curr_head = head
    while curr_head:
        if curr_head.data == prev_data:  # Line 9
            new_node = Node(data = new_data)
            curr_head.next = new_node  # Line 11
            return new_node
        else:
            curr_head = curr_head.next
```

The student incorrectly believes that assigning a new node Y to the `next` field of a node X in a linked list will automatically ensure that Y will be followed by the rest of the list that originally came after X. The reasoning trajectory establishes:

Step A.2: From Test Case 1, before any modifications, the linked list is constructed as: head (data=1) → node (data=2) → node (data=3) → node (data=4). Therefore, initially `head.next.next.next` should refer to the node with data=4.

Step A.3: Test Case 2 calls `insert_after(head, 2, 5)`, which is intended to insert a new node with data=5 after the node with data=2.

Step A.4: Test Case 5 confirms that after the insertion, `head.next.next.data == 5`, meaning the new node with data=5 is successfully positioned at `head.next.next`.

Step A.5: Since the node with data=2 is at `head.next` (Test Case 4), and the new node with data=5 is now at `head.next.next` (A.4), we can confirm that in the code execution, when `curr_head` points to the node with data=2 on line 9, the new node is created and assigned.

The generated teacher question for Step A.5 is: "Let's trace through the code execution. When the condition on line 9 becomes true, which node does `curr_head` point to?"

The LLM judge marked this as invalid, while the human expert marked it as valid. The question effectively prompts the key inference from Step A.5 (identifying which node `curr_head` points to when the condition becomes true). The judge appears to have penalized the conversational preamble "Let's trace through the code execution" as not directly prompting the step. However, the utterance still elicits the correct inference.

C.8.2.2 False Positives: Judge Too Lenient

The judge also incorrectly marked invalid teacher utterances as valid, revealing patterns where the judge aligned too closely with RT structure at the expense of not stating the inference directly in the utterance.

► **Misattributing the Utterance to the Wrong Step.** This case also involves the same `turn_clockwise(compass_point)` buggy code. The reasoning trajectory establishes:

Step A.5: If `compass_point = "N"` were a valid comparison expression (A.4), then placing it in an `if` statement, as in `if compass_point = "N":`, would be grammatically correct Python code. The interpreter would be able to understand and execute it without raising a `SyntaxError`.

Step A.6: However, the interpreter does raise a `SyntaxError` for the code on line 2 (A.1, A.2). This directly contradicts the expectation from Step A.5.

The generated teacher question for Step A.6 is: “But what did the interpreter actually do when it saw that line?”

The LLM judge marked this as invalid, claiming it prompts the wrong RT step, while the human expert marked it as valid. The question appropriately contrasts the student’s expectation (from A.5: if = were valid, the code would parse) with actual interpreter behavior (A.6: interpreter raises `SyntaxError`). The judge appears confused by questions that reference expectations from prior steps, misidentifying this as prompting the wrong step.

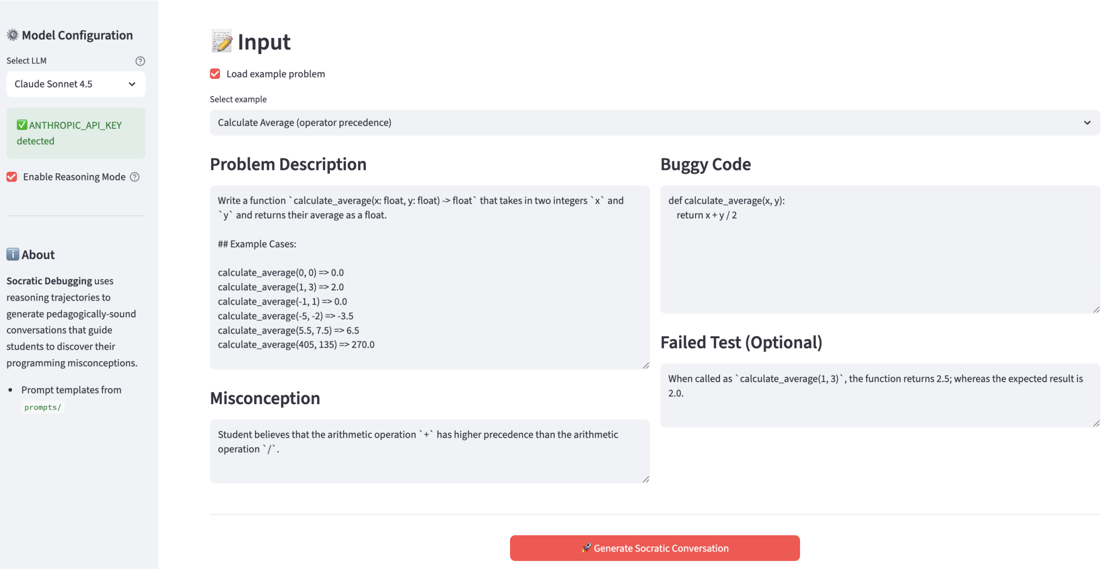
C.8.2.3 Judge Strengths

The judge’s 96.59% overall agreement rate demonstrates strong alignment with human judgment. The judge is particularly reliable on clear-cut cases, including generic opening questions, direct requests for values or expressions, and questions with obvious logical errors. The judge also reliably catches rhetorical questions ending with “right?” that seek confirmation rather than reasoning.

Additionally, the judge appropriately handles implicit information, generally accepting questions where some information is implied. For instance, it recognizes that asking “What does this expression evaluate to?” can expect both the expression and its result, acknowledging that complete pedagogical steps may require multi-part answers.

C.9 Web Interface

To support practical use of our approach, we developed an interactive web application using Streamlit[2] that implements an end-to-end pipeline: from a problem specification and buggy student code to a complete Socratic debugging conversation. The interface provides educators and researchers with immediate access to our methodology without requiring technical expertise or substantial setup.



Socratic Debugging uses reasoning trajectories to generate pedagogically-sound conversations that guide students to discover their programming misconceptions.

- Prompt templates from `prompts/`

Problem Description

```
Write a function `calculate_average(x: float, y: float) -> float` that takes in two integers `x` and
`y` and returns their average as a float.

## Example Cases:

calculate_average(0, 0) => 0.0
calculate_average(1, 3) => 2.0
calculate_average(-1, 1) => 0.0
calculate_average(-5, -2) => -3.5
calculate_average(5.5, 7.5) => 6.5
calculate_average(405, 135) => 270.0
```

Buggy Code

```
def calculate_average(x, y):
  return x + y / 2
```

Failed Test (Optional)

When called as `calculate_average(1, 3)`, the function returns 2.5; whereas the expected result is 2.0.

Misconception

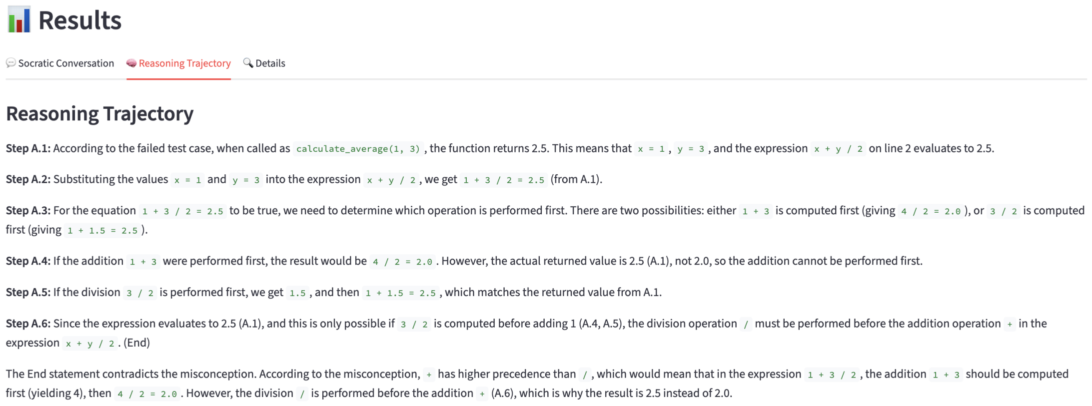
Student believes that the arithmetic operation `+` has higher precedence than the arithmetic operation `/`.

Figure C.4: Interactive web interface for generating reasoning trajectories and Socratic conversations from student code. The interface accepts problem descriptions, buggy implementations, and misconception descriptions as input, and produces structured Socratic intervention.

The workflow begins when users input a problem description along with student code that fails one or more test cases, a description of the failed test case, and a misconception description. The application generates a reasoning trajectory that maps a path from the student's flawed mental model to a contradictory statement about program behavior. This trajectory subsequently guides the generation of a Socratic conversation demonstrating how

---

[2] `https://streamlit.io`



Results

Reasoning Trajectory

Step A.1: According to the failed test case, when called as `calculate_average(1, 3)` , the function returns 2.5. This means that `x = 1` , `y = 3` , and the expression `x + y / 2` on line 2 evaluates to 2.5.

Step A.2: Substituting the values `x = 1` and `y = 3` into the expression `x + y / 2` , we get `1 + 3 / 2 = 2.5` (from A.1).

Step A.3: For the equation `1 + 3 / 2 = 2.5` to be true, we need to determine which operation is performed first. There are two possibilities: either `1 + 3` is computed first (giving `4 / 2 = 2.0`), or `3 / 2` is computed first (giving `1 + 1.5 = 2.5`).

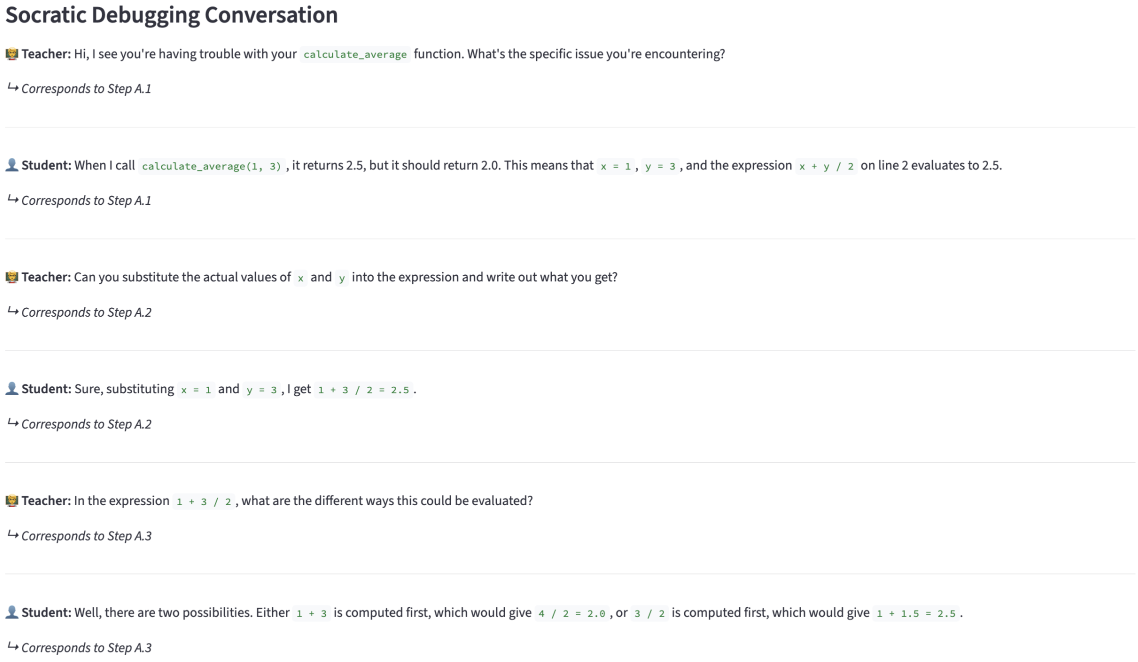
Step A.4: If the addition `1 + 3` were performed first, the result would be `4 / 2 = 2.0` . However, the actual returned value is 2.5 (A.1), not 2.0, so the addition cannot be performed first.

Step A.5: If the division `3 / 2` is performed first, we get `1.5` , and then `1 + 1.5 = 2.5` , which matches the returned value from A.1.

Step A.6: Since the expression evaluates to 2.5 (A.1), and this is only possible if `3 / 2` is computed before adding 1 (A.4, A.5), the division operation `/` must be performed before the addition operation `+` in the expression `x + y / 2` . (End)

The End statement contradicts the misconception. According to the misconception, `+` has higher precedence than `/` , which would mean that in the expression `1 + 3 / 2` , the addition `1 + 3` should be computed first (yielding 4), then `4 / 2 = 2.0` . However, the division `/` is performed before the addition `+` (A.6), which is why the result is 2.5 instead of 2.0.

Figure C.5: Interactive web interface generates a reasoning trajectory concluding with a statement that contradicts the misconception of addition having a higher precedence than division.



Socratic Debugging Conversation

Teacher: Hi, I see you're having trouble with your `calculate_average` function. What's the specific issue you're encountering?

↳ *Corresponds to Step A.1*

Student: When I call `calculate_average(1, 3)` , it returns 2.5, but it should return 2.0. This means that `x = 1` , `y = 3` , and the expression `x + y / 2` on line 2 evaluates to 2.5.

↳ *Corresponds to Step A.1*

Teacher: Can you substitute the actual values of `x` and `y` into the expression and write out what you get?

↳ *Corresponds to Step A.2*

Student: Sure, substituting `x = 1` and `y = 3` , I get `1 + 3 / 2 = 2.5` .

↳ *Corresponds to Step A.2*

Teacher: In the expression `1 + 3 / 2` , what are the different ways this could be evaluated?

↳ *Corresponds to Step A.3*

Student: Well, there are two possibilities. Either `1 + 3` is computed first, which would give `4 / 2 = 2.0` , or `3 / 2` is computed first, which would give `1 + 1.5 = 2.5` .

↳ *Corresponds to Step A.3*

Figure C.6: The tool generates a complete Socratic conversation between a student and a teacher based on the generated reasoning trajectory.

an instructor might guide the student toward self-discovery of their error.

Users can select from multiple state-of-the-art LLMs (Claude, GPT, Gemini), with the interface automatically applying the same prompt templates and model configurations used in our experimental evaluation. Reasoning capabilities are enabled by default for all compatible models. The interface presents results in a structured format featuring: the complete reasoning trajectory with intermediate steps, an example Socratic conversation, and expandable reasoning traces showing the model's internal deliberation process.

For classroom applications, instructors can use this tool during office hours or help sessions to rapidly prepare targeted interventions for individual students. When a student presents buggy code, the instructor can input it into the system and within seconds receive a principled Socratic questioning strategy tailored to a specific misconception. This enables more effective one-on-one debugging sessions by providing the instructor with a structured pedagogical plan rather than ad-hoc questioning. The application runs locally and securely loads API credentials from environment variables.