McMining: Automated Discovery of Misconceptions in Student Code

Erfan Al-Hossami and Razvan Bunescu
University of North Carolina at Charlotte
Charlotte, NC, USA
{ealhossa, rbunescu}@charlotte.edu

Abstract

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. In this paper, we introduce 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.
https://github.com/taisazero/mcminer

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 1.

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 (Qian and Lehman, 2017) encountered

➢ **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`.

➢ **Student code**

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

➢ **Potential misconception**:

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

Figure 1: Problem-code pair that exhibits a *potential* programming misconception about the `range` function.

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 (Robins, 2010). 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 (Al-Hossami et al., 2024). However, this process can be cognitively demanding, and, due to insufficient TA resources (Yadav et al., 2016), not fast enough to keep up with the volume of concepts introduced in a course.

Recognizing the importance of identifying misconceptions early, in this paper we introduce McMining, the novel task of 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). 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 4). We present positive results of the two version of the McMiner tool on the benchmark dataset, using different LLMs (Section 5). The paper ends with conclusions and thoughts on future work (Section 6).

2 Related Work

Traditional automated systems (Johnson and Soloway, 1984; Sirkia and Sorva, 2012; Evans et al., 2023) rely on hand-crafted rules targeting a specific set of misconceptions, consequently they are limited to identifying only misconceptions from this predefined set. For example, when run on the example from Figure 1, Side-lib (Evans et al., 2023) does not identify any misconception. Similarly, modern LLM-based approaches such as (Lee et al., 2024) 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 (Brown and Altadmri, 2014). In parallel, other approaches aim to identify code segments that contain logical errors, such as (Mens et al., 2021; Hoq et al., 2025), *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 (Liu et al., 2024) aim to identify common coding errors, style inconsistencies, or security vulnerabilities through automated code inspection. Static analysis fundamentally differs from misconception mining in both scope and objective. Static analysis operates on individual code samples to identify coding issues, which limits their ability to determine whether an otherwise syntactically correct code matches a problem specification. When run on the code from Figure 1, for example, PythonTA only provides code-styling suggestions. In contrast, the misconception mining task introduced in this paper analyzes patterns across multiple student code samples in order to identify underlying false beliefs about programming language constructs. Critically, not all misconceptions manifest as coding errors, as a student may hold a false belief that happens to produce correct output on available test cases, and conversely, not all coding errors indicate misconceptions, as students may make coding mistakes, such as typos, that are not caused by false beliefs. Our work focuses on mining the conceptual misunderstandings that cause coding errors, rather than detecting the errors themselves, enabling targeted pedagogical interventions that address the root cause rather than symptoms.

3 Task Definition and Benchmark Dataset

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):

$$PC = \{(p_1,c_1),(p_2,c_2),...,(p_N,c_N)\} = \{(p_n,c_n)\}_1^N \quad (1)$$

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 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 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

Algorithm 1 Benchmark Dataset Generation

```
1:  Initialize dataset $\mathcal{D} \leftarrow \emptyset$
2:  for each student $k \in \{1,2,...,K\}$ 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
```

disjoint categories, *harmful* vs. *benign*, depending on whether they cause or not a bug in the code.

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 $mc^{(k)}$. More formally, the dataset contains the following components:

$$\mathcal{D} = \left\{ \left(PC^{(k)}, mc^{(k)} \right) \right\}_1^K ; \quad PC^{(k)} = \left\{ \left(p_n^{(k)}, c_n^{(k)} \right) \right\}_1^{N_k} \tag{2}$$

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 $\mathcal{MC} = \{mc^{(m)}\}_1^M$ 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 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 3.1 below we describe the McInject tool that we developed for this purpose.

Table 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.

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 1: Benchmark dataset statistics.

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.

3.1 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 E provides further detailed descriptions of the tool development, including the prompt, LLM hyper-parameters, and evaluation setting.

4 The McMiner-S and McMiner-M Tools

As defined in Section 3, 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

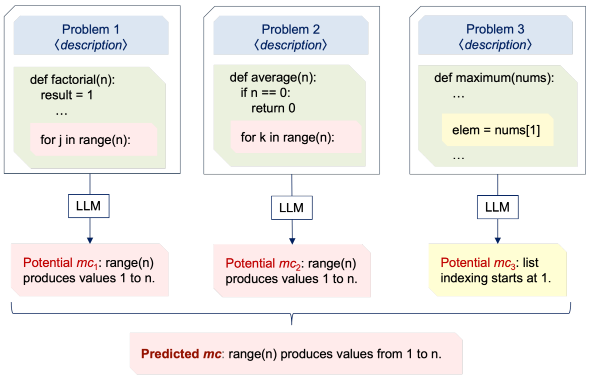


Figure 2: Illustration of the McMiner-S single-instance mining approach on a bag with 3 problem-code pairs.

Algorithm 2 McMiner-S

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

```
1: Initialize set of misconceptions $M \leftarrow \emptyset$
2: **for** each problem-code pair $(p_j,c_j) \in PC$ **do**
3:     Identify potential misconception $mc_j$ for $(p_j,c_j)$
4:     If $mc_j$ found, add to $M$
5: **if** $M$ is empty **then**                 ▷ no misconception found
6:     **return** $\epsilon$
7: **for** each misconception $mc \in M$ **do**
8:     $count(mc) = |\{(p_j,c_j) \in PC | mc_j = mc\}|$
9: **return** $\hat{mc} = \operatorname{argmax}_{mc \in M} count(mc)$
```

two variants of an LLM-based mining tool:

1. A *single instance* variant McMiner-S
2. A *multiple instance* variant McMiner-M.

McMiner-S proceeds in two stages, as illustrated in Figure 2 on a bag with 3 problem-code pairs, and further detailed in Algorithm 2. First (steps 2 to 4), an LLM is instructed to identify potential misconceptions from each problem-code pair, using the prompt shown in Appendix F. Then (steps 5 to 10), the most frequently occurring misconception is returned.

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

5 Experimental Evaluations

We used the benchmark dataset to evaluate the effectiveness of the two McMiner tools when instantiated using three large LMs: 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 H. For computing accuracy, precision, recall, F1-score, we define: *true positives* as cases where the Ground Truth (GT) misconception matches the prediction, or the prediction is a validation novel misconception; *true negatives* as cases where both GT and prediction contain no misconception; *false positives* as cases where the prediction does not match GT and fails validation; and *false negatives* as 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).

To determine whether predicted misconceptions semantically match ground truth misconceptions, we employed an LLM-as-judge approach. Given that Claude-Sonnet-4.5 was already used in McInject, to mitigate any effects due to potential self-enhancement bias (Zheng et al., 2023), we use GPT-5 with medium reasoning effort as LLM-as-judge for all evaluations. Additional evaluation results using three different LLMs-as-judges are presented in Appendix I, together with pairwise agreement results that show over 94% agreement, indicating that the LLM-as-judge approach is reliable for this task. To credit discovery of misconceptions 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.

Table 2 present the performance of the 3 large models in the multiple and single instance mining settings. Multi-instance mining (McMiner-M) substantially outperforms single-instance mining (McMiner-S), with the best model (Claude Sonnet-4.5 + Reasoning) achieving 81.4% accuracy for McMiner-M compared to 77.7% 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 (77.8% to 81.4%), Gemini (60.4% to 76.1%), and

	McMiner-M				McMiner-S
Large models	**Precision**	**Recall**	**F1**	**Accuracy**	**Accuracy**
OpenAI o3-mini (low-effort)	84.3%	67.9%	75.2%	74.6%	76.6%
OpenAI o3-mini (medium-effort)	82.4%	71.8%	76.8%	76.4%	**77.7%**
Anthropic Claude Sonnet-4.5	78.1%	76.5%	77.2%	77.8%	64.5%
Anthropic Claude Sonnet-4.5 + Reasoning	83.0%	78.1%	**80.5%**	**81.4%**	69.0%
Gemini 2.5-flash	78.8%	56.7%	65.9%	60.4%	73.9%
Gemini 2.5-flash + Reasoning	76.0%	76.3%	76.1%	76.1%	69.1%

Table 2: McMiner results for multiple-instance mining (left) and single-instance mining (right), evaluated using GPT-5 (medium reasoning effort) as judge.

	McMiner-M			
Small models	**P**	**R**	**F1**	**Acc**
Qwen-3-8B	75.9%	43.1%	55.0%	53.7%
Qwen-3-8B + R	73.0%	51.0%	60.0%	57.5%
Qwen-3-14B	81.5%	59.1%	68.5%	69.0%
Qwen-3-14B + R	70.2%	60.2%	64.9%	59.3%

Table 3: McMiner-M results using Qwen-3 models.

o3-mini (74.6% to 76.4% with medium effort).

Additionally, in Table 3 we present the results of McMiner-M when using the Qwen-3-8B and Qwen-3-14B models. The results show that the smaller models obtain competitive performance, with Qwen-3-14B achieving 69.0% accuracy, demonstrating the feasibility of misconception mining with open source models as well.

5.1 Error Analysis

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. However, bags that contain only correct code samples could be easily filtered out by evaluating on the corresponding test cases, which means that in practice the system would still be highly useful for the identification of harmful misconceptions.

Overall, McMiner is also effective at identifying novel misconceptions: Claude Sonnet-4.5 with reasoning discovered 41 novel true positives (12.1% of bags), for example *"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 over 95% accuracy, while subtle benign misconceptions, e.g., unnecessary parentheses, and subtle out-of-distribution cases, such as vowel-based naming, prove more challenging, especially for single-instance mining. Misconceptions that require the LLM to reason from the student's perspective, e.g., operator precedence, achieve between 42% – 50% accuracy. Overall, compared with the single-instance version, multi-instance mining obtains improved performance on the more difficult cases.

6 Conclusion

This paper 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 81.4% 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, which can enable more personalized learning interventions.

Future work includes systematic evaluations of McMiner on real student submissions and studying the effectiveness of its pedagogical integration in a classroom setting.

Acknowledgments

This research project has benefited from the Microsoft Accelerating Foundation Models Research (AFMR) grant program. The research was also partly supported by the United States Air Force (USAF) under Contract No. FA8750-21-C-0075. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the USAF.

Limitations

The McMining benchmark dataset contains code samples that exhibit a single primary misconception, while code from real students may exhibit multiple misconceptions simultaneously. In general, the distribution of misconceptions in the dataset may be different from their real-world distribution. The evaluation is limited to Python, as such generalization to other programming languages remains to be explored. While the misconception dataset contains 67 initial misconceptions and 42 novel misconceptions discovered by LLMs (which to our knowledge is the largest ever), the dataset is expected to be incomplete, as the complete set of student misconceptions is virtually unbounded.

The LLM-based approach incurs computational costs that may limit large-scale deployment in resource-constrained educational environments. Furthermore, over-reliance on automated detection may reduce instructors' diagnostic skills and discourage valuable Socratic dialogue with students. Instructors using the system may risk premature intervention before students have the opportunity for self-correction through debugging, a critical part of learning to code. Additionally, processing student code through third-party LLM APIs may raise data privacy concerns and requires careful consideration of educational data protection regulations, such as FERPA.

References

Erfan Al-Hossami, Razvan Bunescu, Justin Smith, and Ryan Teehan. 2024. 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, page 53–59, New York, NY, USA. Association for Computing Machinery.

Erfan Al-Hossami, Razvan Bunescu, Ryan Teehan, Laurel Powell, Khyati Mahajan, and Mohsen Dorodchi. 2023. 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)*, pages 709–726, Toronto, Canada. Association for Computational Linguistics.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and 1 others. 2021. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*.

Neil CC Brown and Amjad Altadmri. 2014. Investigating novice programming mistakes: Educator beliefs vs student data. In *Proceedings of the 2014 conference on Innovation & technology in computer science education*, pages 43–50.

Luca Chiodini, Igor Moreno Santos, Andrea Gallidabino, Anya Tafliovich, André L. Santos, and Matthias Hauswirth. 2021. 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, page 380–386, New York, NY, USA. Association for Computing Machinery.

Adrian de Freitas, Joel Coffman, Michelle de Freitas, Justin Wilson, and Troy Weingart. 2023. 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, page 938–944, New York, NY, USA. Association for Computing Machinery.

Andrew Ettles, Andrew Luxton-Reilly, and Paul Denny. 2018. Common logic errors made by novice programmers. In *Proceedings of the 20th Australasian Computing Education Conference*, ACE '18, page 83–89, New York, NY, USA. Association for Computing Machinery.

Abigail Evans, Zihan Wang, Jieren Liu, and Mingming Zheng. 2023. 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, page 159–165, New York, NY, USA. Association for Computing Machinery.

Muntasir Hoq, Ananya Rao, Reisha Jaishankar, Krish Piryani, Nithya Janapati, Jessica Vandenberg, Bradford Mott, Narges Norouzi, James Lester, and Bita Akram. 2025. Automated Identification of Logical Errors in Programs: Advancing Scalable Analysis of Student Misconceptions. pages 90–103.

W Lewis Johnson and Elliot Soloway. 1984. Proust: An automatic debugger for Pascal programs. In *Proceedings of the National Conference on Artificial Intelligence*, pages 181–184.

Yanggyu Lee, Suchae Jeong, and Jihie Kim. 2024. 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*, pages 91–103, Berlin, Heidelberg. Springer-Verlag.

David Liu, Jonathan Calver, and Michelle Craig. 2024. A static analysis tool in CS1: Student usage and perceptions of PythonTA. In *Proceedings of the 26th Australasian Computing Education Conference*, ACE '24, pages 172–181, New York, NY, USA. Association for Computing Machinery.

Kim Mens, Siegfried Nijssen, and Hoang-Son Pham. 2021. 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)*, pages 1–8. ACM.

Yizhou Qian and James Lehman. 2017. Students' Misconceptions and Other Difficulties in Introductory Programming: A Literature Review. *ACM Trans. Comput. Educ.*, 18(1):1:1–1:24.

Anthony Robins. 2010. Learning edge momentum: A new account of outcomes in CS1. *Computer Science Education*, 20(1):37–71. Publisher: Routledge _eprint: https://doi.org/10.1080/08993401003612167.

Teemu Sirkia and Juha Sorva. 2012. 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*, pages 19–28.

Aman Yadav, Sarah Gretter, Susanne Hambrusch, and Phil Sands. 2016. Expanding computer science education in schools: Understanding teacher experiences and challenges. *Computer Science Education*, 26(4):235–254.

Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric P. Xing, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. 2023. Judging LLM-as-a-judge with MT-bench and Chatbot Arena. In *Proceedings of the 37th International Conference on Neural Information Processing Systems*, NIPS '23, Red Hook, NY, USA. Curran Associates Inc.

A McInject Qualitative Analysis

We conducted a qualitative analysis of MCINJECT's generated code samples to understand performance variation across misconception types.

A.1 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 misconception similarly produces code with an unmistakable incorrect pattern.

A.2 Failure Patterns

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

A.2.1 Ambiguous Correctness

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 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`. McInject tends to focus on emulating the code structure rather than writing code that the student with the misconception would write and genuinely believes that it would solve the problem correctly.

A.2.2 Unnatural Code

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.

B McMiner Analysis

In this section, we examine the performance of McMiner on the benchmark dataset.

B.1 Performance on Correct-Only vs. Misconception Bags

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 4.

Model	**Reasoning**	**Correct-Only Accuracy**	**Misc. Bags Accuracy**
Claude Sonnet 4.5	✓	59.30%	89.72%
Claude Sonnet 4.5	×	41.86%	91.30%
OpenAI o3-mini (medium)	–	66.28%	80.63%
OpenAI o3-mini (low)	–	75.58%	75.49%
Gemini 2.5 Flash	✓	36.05%	90.91%
Gemini 2.5 Flash	×	61.63%	60.87%

Table 4: 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).

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%).

B.1.1 Common False Positive Patterns

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.

B.2 Novel Misconception Discovery

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 K), 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%).

B.2.1 Cherry-Picked Novel Discovery

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.

B.3 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 functional bugs but instead reflect stylistic preferences or suboptimal practices.

Low-Performing Misconceptions:

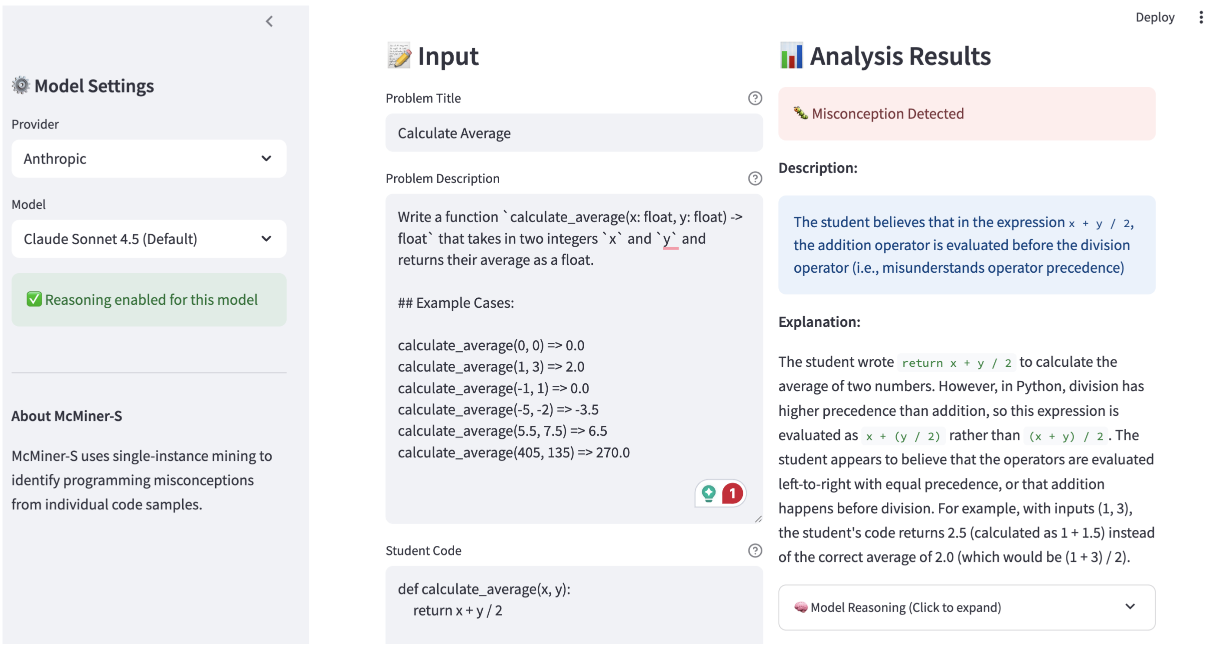
- "Student believes that the `return` statement requires parentheses around its argument" code like `return(x + y)` is syntactically valid and functions correctly
- "Student believes that function parameters automatically change in recursive calls without explicit modification.". This is a harmful misconception 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 definitively determine whether `return(x)` reflects a misconception or a stylistic preference influenced by other programming languages

C McMiner Interface

To facilitate the use of McMiner for educators and researchers, we developed an interactive web application using the Streamlit[1] 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 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.



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

```python
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).

Model Reasoning (Click to expand)

Figure 3: 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.

D Dataset Sources

Our misconceptions were adapted from: The Python Misconception bank (Chiodini et al., 2021) (17 misconceptions), SIDELib (Evans et al., 2023) (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

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

misconceptions, our benchmark contains 1,063 code samples exhibiting those misconceptions. Our benchmark can be easily extended given any new misconception.

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 (Austin et al., 2021), 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 (Al-Hossami et al., 2023, 2024), which focuses on debugging scenarios, along with 8 problems from the Auckland dataset (Ettles et al., 2018) and 6 from the FalconCode dataset (de Freitas et al., 2023). Finally, 19 problems were handwritten by the authors, drawing inspiration from public programming courses and educational websites such as w3resource to fill gaps in programming language construct coverage.

Table 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 4), 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.

D.1 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 4 shows the writing guidelines used to create the misconception bank.

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 4: Misconception description writing guidelines.

E 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 5.

To further improve code quality, we implement an iterative refinement process. After McInject generates code, an LLM-as-a-judge evaluates whether the code properly exhibits the target misconception (see Section K.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.

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 5: Prompt template for McInject tool.

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 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-a-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-a-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-a-judge approach in determining whether a code sample exhibits a misconception description. The LLM-as-a-judge leveraged Claude-Sonnet-4.5 with the same hyperparameters as McInject.

F The McMiner-S Prompt Template

This section describes the prompt used by the McMiner-S tool.

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

G The McMiner-M Prompt Template

This section describes the prompt used by the McMiner-M tool.

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 7: Prompt template for MCMINER-M (multi-instance mining). The full template includes additional metadata fields and guidelines.

H Experimental Setup

We evaluated MCMINER using five LLMs, comprising three state-of-the-art commercial models and two open-source models:

- **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 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, 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.
- **Qwen-3**: We evaluated two variants of the Qwen-3 model (8B and 14B parameters) to assess performance of smaller, more resource-efficient models. For non-reasoning mode, we used temperature 0.7, top_p 0.8, top_k 20, and max_tokens 4000. For reasoning mode, we used temperature 0.6, top_p 0.95, top_k 20, and max_tokens 4000. These hyperparameters follow the recommendations from the official Qwen-3 repository.

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.

I LLM-as-Judge Robustness Analysis

To address potential concerns about self-enhancement bias (Zheng et al., 2023) when using Claude-Sonnet-4.5 (the same model was used in MCINJECT) as the judge, we conducted a robustness analysis using three independent LLM judges: Claude Sonnet-4.5 with reasoning, GPT-5 with medium reasoning effort, and Gemini-2.5-Pro. This section presents the complete results from all three judges and quantifies their agreement.

I.1 Results with Claude Sonnet-4.5 as Judge

Table 5 shows the evaluation results using Claude Sonnet-4.5 with reasoning as the judge.

	MCMINER-M				MCMINER-S
Language Model	**Prec.**	**Rec.**	**F1**	**Acc.**	**Acc.**
OpenAI o3-mini (low)	85.6%	67.5%	75.5%	75.5%	76.9%
OpenAI o3-mini (medium)	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 + Reasoning	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 + Reasoning	77.0%	76.7%	76.8%	76.9%	69.4%

Table 5: MCMINER results evaluated using Claude Sonnet-4.5 with reasoning as judge.

I.2 Results with GPT-5 as Judge

Table 6 presents results using GPT-5 with medium reasoning effort as an independent judge. These results are reported in the main paper (Table 2).

	MCMINER-M				MCMINER-S
Language Model	**Prec.**	**Rec.**	**F1**	**Acc.**	**Acc.**
OpenAI o3-mini (low)	84.3%	67.9%	75.2%	74.6%	76.6%
OpenAI o3-mini (medium)	82.4%	71.8%	76.8%	76.4%	77.7%
Claude Sonnet-4.5	78.1%	76.5%	77.2%	77.8%	64.5%
Claude Sonnet-4.5 + Reasoning	83.0%	78.1%	80.5%	81.4%	69.0%
Gemini 2.5-flash	78.8%	56.7%	65.9%	60.4%	73.9%
Gemini 2.5-flash + Reasoning	76.0%	76.3%	76.1%	76.1%	69.1%

Table 6: MCMINER results evaluated using GPT-5 (medium reasoning) as judge.

I.3 Results with Gemini-2.5-Pro as Judge

Table 7 shows results using Gemini-2.5-Pro.

	MCMINER-M				MCMINER-S
Language Model	**Prec.**	**Rec.**	**F1**	**Acc.**	**Acc.**
OpenAI o3-mini (low)	85.6%	69.2%	76.5%	75.5%	76.9%
OpenAI o3-mini (medium)	83.2%	73.4%	78.0%	76.9%	78.1%
Claude Sonnet-4.5	80.8%	80.2%	80.5%	80.2%	67.1%
Claude Sonnet-4.5 + Reasoning	84.1%	81.1%	82.6%	82.3%	69.8%
Gemini 2.5-flash	79.8%	57.2%	66.6%	61.1%	72.5%
Gemini 2.5-flash + Reasoning	80.0%	79.4%	79.7%	79.6%	69.4%

Table 7: MCMINER results evaluated using Gemini-2.5-Pro as judge.

I.4 Inter-Judge Agreement Analysis

We computed pairwise agreement between all three LLM judges. For each bag in our dataset of 339 bags and each of the 12 experimental configurations (6 models $\times$ 2 McMiner variants), we determined whether each judge classified the prediction as: (1) matching ground truth, (2) not matching ground truth, or (3) a valid novel misconception.

The agreement analysis reveals high consistency across judges with over 94% agreement across all pairs. This high inter-judge agreement demonstrates that our evaluation results are robust to the choice of judge and that the self-enhancement bias concern is minimal. The consistency holds even when evaluating novel misconceptions, where judges must assess whether a prediction that doesn't match ground truth still represents a valid programming misconception exhibited in the code.

J Single Problem-Code Pair Analysis

While the main experiments in this paper evaluate McMiner on bags containing 4-8 problem-code pairs, a common scenario in educational settings is analyzing a single problem-code pair from a student. To understand performance in this setting, we conducted an additional experiment where we evaluate models on individual problem-code pairs from our dataset.

For this experiment, we randomly sampled one problem-code pair from each of the 339 bags in our benchmark dataset and applied the McMiner-S prompt to each sample. We used GPT-5 with medium reasoning effort as the judge, consistent with the main paper evaluation. Table 8 presents the accuracy results.

Language Model	**Accuracy**
OpenAI o3-mini (low-effort)	51.8%
OpenAI o3-mini (medium-effort)	53.3%
Anthropic Claude Sonnet-4.5	60.8%
Anthropic Claude Sonnet-4.5 + Reasoning	60.1%
Gemini 2.5-flash	53.3%
Qwen-3-8B	38.6%
Qwen-3-8B + Reasoning	42.3%
Qwen-3-14B	43.8%
Qwen-3-14B + Reasoning	46.9%

Table 8: McMiner accuracy on single problem-code pairs, evaluated using GPT-5 (medium reasoning) as judge.

The results reveal a substantial performance gap between single-pair and multi-pair settings. The best performing model, Claude Sonnet-4.5, achieves 60.8% accuracy on single pairs compared to 77.8% on bags of 4-8 pairs (Table 2). This performance degradation is expected and aligns with the core motivation of our multi-instance approach: analyzing a single code sample often provides insufficient evidence to confidently identify a misconception, as the same bug could arise from multiple different misconceptions or simply be a typo. Observing this pattern across multiple code samples significantly increases confidence that it reflects a genuine misconception. These results underscore the value of the multi-instance learning formulation for misconception mining, where having multiple code samples enables pattern recognition and reduces ambiguity.

K Evaluation Prompts

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

K.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 8 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 8: Prompt template for LLM-as-a-judge evaluation of MCINJECT generated code samples. The full prompt template includes metadata fields to output such as rationale and confidence level.

K.2 McMiner Semantic Matching Prompt

To evaluate MCMINER predictions, we use an LLM-as-a-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 9 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 9: Prompt template for semantic matching evaluation of McMiner predictions. The full prompt template includes metadata fields to output such as confidence level and explanation.