GuardRails: Automated Suggestions for Clarifying Ambiguous Purpose Statements

Mrigank Pawagi
Indian Institute of Science
Bengaluru, Karnataka, India
mrigankp@iisc.ac.in

Viraj Kumar
Indian Institute of Science
Bengaluru, Karnataka, India
viraj@iisc.ac.in

ABSTRACT

Before implementing a function, programmers are encouraged to write a *purpose statement* i.e., a short, natural-language explanation of what the function computes. A purpose statement may be *ambiguous* i.e., it may fail to specify the intended behaviour when two or more inequivalent computations are plausible on certain inputs. Our paper makes four contributions. First, we propose a novel heuristic that suggests such inputs using Large Language Models (LLMs). Using these suggestions, the programmer may choose to clarify the purpose statement (e.g., by providing a *functional example* that specifies the intended behaviour on such an input). Second, to assess the quality of inputs suggested by our heuristic, and to facilitate future research, we create an open dataset of purpose statements with known ambiguities. Third, we compare our heuristic against GitHub Copilot's Chat feature, which can suggest similar inputs when prompted to generate unit tests. Fourth, we provide an open-source implementation of our heuristic as an extension to Visual Studio Code for the Python programming language, where purpose statements and functional examples are specified as docstrings and doctests respectively. We believe that this tool will be particularly helpful to novice programmers and instructors.

CCS CONCEPTS

• **Applied computing** → *Computer-assisted instruction*; • **Social and professional topics** → **Student assessment**.

KEYWORDS

function design, purpose statement, CS1

ACM Reference Format:
Mrigank Pawagi and Viraj Kumar. 2023. GuardRails: Automated Suggestions for Clarifying Ambiguous Purpose Statements. In *16th Annual ACM India Compute Conference (COMPUTE '23), December 09–11, 2023, Hyderabad, India.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3627217.3627234

1 INTRODUCTION

Large Language Models (LLMs) can generate code from natural language prompts [5]. Although this code is not always accurate, an early study showed that LLM-generated code outperforms novice programmers on simple code-writing tasks [10]. More recent work shows continued improvement, including on more complex programming tasks [13, 17]. As a result, there have been calls to urgently review "our educational practices in the light of these new technologies" [4]. One such review of code-writing tasks has been put forward by Raman and Kumar [21], based on the 6-step recipe proposed by Felleisen et al. [8] to help novice programmers design functions. We focus on two of these steps:

Step 2 (Signature, Purpose Statement, Header) State what kind of data the desired function consumes and produces. Formulate a concise answer to the question *what* the function computes. Define a stub that lives up to the signature.

Step 3 (Functional Examples) Work through examples that illustrate the function's purpose.

1.1 Motivating Example

In the following Python code, the signature (Line 1) includes a meaningful function name and type-hints[1] for its argument and return type. Further, the purpose statement (Line 2) is expressed as a *docstring*, with one functional example as a *doctest*[2] (Lines 4-5):

```
1  def first_nonzero(nums: list[float]) -> float:
2      """Return the first non-zero value in nums.
3  
4      >>> first_nonzero([0.0, 3.7, 0.0])
5      3.7
6      """
```

This purpose statement is ambiguous. For the class of inputs containing no non-zero values (e.g., `nums = []`), it is unclear what the function should do. When we used GitHub Copilot to generate multiple solutions with the above prompt, it resolved this ambiguity in three ways[3] (see Figure 2). The first implementation raises an error when `nums` contains no non-zero values[4]:

```
7      for num in nums:
8          if num != 0.0:
9              return num
10     raise ValueError("No non-zero numbers in the
   ↪ list")
```

The second implementation resolves the ambiguity differently, by replacing Line 10 of the above solution with `return 0.0`. Our tool, GuardRails, cannot determine which of these (if any) is the

[1]https://docs.python.org/3/library/typing.html
[2]https://docs.python.org/3/library/doctest.html
[3]LLMs are usually not deterministic [5] and can generate different solutions for the same prompt when executed multiple times.
[4]The third implementation raises a different type of error on such inputs.

intended behaviour, but it can at least alert the programmer about this ambiguity by suggesting the input `[]`[5]. Such an input can also be suggested by Copilot's Chat feature, by prompting it to generate tests[6] for this function (Figure 1). Notice that these tests are based on an *assumption* that the expected return value is `None`. Copilot Chat does not explicitly draw attention to potential ambiguities by highlighting these assumptions, so it is up to the programmer to recognize the presence of assumptions.

If this was the *only* input suggested to the programmer, they might attempt to clarify the purpose statement by including an assumption: `nums` will contain at least one non-zero value. However, our tool (but not Copilot Chat) identifies an example from a second class of inputs that exposes a subtler ambiguity: `[nan]`. This list contains "not a number", an easy-to-overlook non-zero value whose special properties ensure that the two implementations are once again inequivalent on this input (since `nan != nan` [1]). Using GuardRails' suggestions, the programmer might clarify ambiguities in the original purpose statement as shown below (Lines 2-4). Lines 9-12 show GitHub Copilot's first suggested implementation.

```
1  def first_nonzero(nums: list[float]) -> float:
2      """Return the first non-zero value,
3      excluding NaN, in nums. If no such
4      value exists, return 0.0.
5  
6      >>> first_nonzero([0.0, 3.7, 0.0])
7      3.7
8      """
9      for num in nums:
10         if num != 0.0 and not math.isnan(num):
11             return num
12     return 0.0
```

Our implementation of GuardRails is based on a heuristic, which we detail in Section 3.

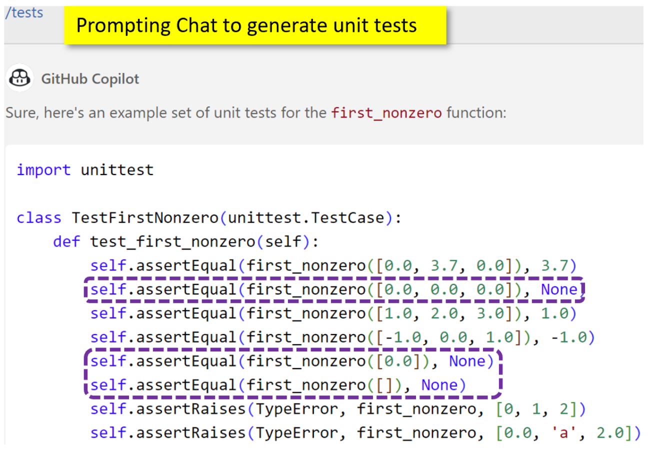
1.2 Research Questions

To evaluate our heuristic, we have created a dataset of 15 functions, each with between 1 and 3 ambiguous input classes (AICs). The original version of the `first_nonzero()` function belongs to this dataset, and it has two AICs: "Non non-zero numbers" and "NaN as the only non-zero number". Unlike our heuristic, Copilot Chat does not explicitly highlight certain inputs as potentially ambiguous. Nevertheless, since GitHub Copilot uses a state-of-the-art LLM and is increasingly being adopted by the professional software development community[7], we believe that it provides a good benchmark for comparison with our tool. We first compare the abilities of Copilot Chat and GuardRails across multiple problems:

RQ1 In relative terms, how do the abilities of both tools to suggest inputs from known AICs vary across the functions in our dataset?

Further, since LLMs are sensitive to the level of detail provided, we expect both these LLM-based techniques to leverage additional

[5]Our tool attempts to report the *simplest* such input.
[6]https://docs.github.com/en/copilot/github-copilot-chat/about-github-copilot-chat#generating-unit-test-cases
[7]https://github.blog/2023-06-27-the-economic-impact-of-the-ai-powered-developer-lifecycle-and-lessons-from-github-copilot/


/tests Prompting Chat to generate unit tests

GitHub Copilot

Sure, here's an example set of unit tests for the `first_nonzero` function:

```
import unittest

class TestFirstNonzero(unittest.TestCase):
    def test_first_nonzero(self):
        self.assertEqual(first_nonzero([0.0, 3.7, 0.0]), 3.7)
        self.assertEqual(first_nonzero([0.0, 0.0, 0.0]), None)
        self.assertEqual(first_nonzero([1.0, 2.0, 3.0]), 1.0)
        self.assertEqual(first_nonzero([-1.0, 0.0, 1.0]), -1.0)
        self.assertEqual(first_nonzero([0.0]), None)
        self.assertEqual(first_nonzero([]), None)
        self.assertRaises(TypeError, first_nonzero, [0, 1, 2])
        self.assertRaises(TypeError, first_nonzero, [0.0, 'a', 2.0])
```

Figure 1: When GitHub Copilot Chat is prompted to generate unit tests, it suggests examples from only one of the two Ambiguous Input Classes (AICs) for this function. For each of these examples (highlighted), Copilot Chat *assumes* that the return value is `None`.

details to identify ambiguities in purpose statement. We consider four variants of each function, with progressively increasing amounts of detail:

S Only the function's *Signature*.
SP In addition to **S**, the (ambiguous) *Purpose statement*.
SP1 In addition to **SP**, *one* functional example that does not correspond to any AIC.
SPx In addition to **SP1**, one or more functional examples that further *explore* the input space, but again do not correspond to any AIC.

The original version of the `first_nonzero()` function presented in this paper is the **SP1** variant with one functional example for the list `[0.0, 3.7, 0.0]`. The **SPx** variant adds a functional example for the list `[-3.14]`, which explores parts of the input space that include negative numbers and are "closer" to the empty list. We included **SPx** variants in our study after realising that LLMs can be prompted to generate a richer variety of solutions when such inputs are included. Our second research question is:

RQ2 In absolute terms, does the percentage of inputs from known AICs increase as we move from **S** to **SPx** variants?

2 RELATED WORK

Real-world problems are often poorly specified, and the failure of programmers to clarify crucial details before writing code is a known root cause of software project failure [6]. Although instructors in CS1 courses often assist novice programmers by providing unrealistically detailed problem specifications, prior work has established that some students fail to comprehend these details and are less likely to produce correct code [20, 24]. In contrast, we echo Schneider's view [23] that CS1 students need greater exposure to problem specifications that are realistic in the sense of containing ambiguities or lacking key details. We believe that such exposure is particularly relevant at a time when LLMs such as Codex [5] can outperform a majority of students [10] on CS1 tasks [7, 26].

While several approaches have been proposed to help students comprehend well-specified problems [14, 20, 27, 28], including approaches suitable for linguistically diverse countries such as India [2], we are unaware of prior work that explicitly points out ambiguities in a given problem. As noted previously, Copilot Chat can implicitly indicate the presence of ambiguities by generating unit tests based on *unacknowledged* assumptions. An aspect of our approach is similar to the Test-Driven User-Intent Formalization workflow proposed by Lahiri et al. [11]. Here, the programmer seeks to clarify their intent by providing functional examples, and the LLM is *constrained* to generate code that satisfies these examples. GuardRails similarly uses functional examples to filter out certain implementations (see Section 3).

LLMs have been used to support CS education in a variety of ways, including explaining code [16], creating code-writing tasks [16, 22], providing students with hints on such tasks [18], including how to fix syntactic and logical errors [3, 12] . It is important to note that the output of LLMs is not always correct. For example, LLMs can generate incorrect code [7] and incorrect explanations [3, 22].

To the best of our knowledge, the approach used by Copilot Chat to generate unit tests uses *only* an appropriately trained LLM. In contrast, our approach uses LLMs to generate multiple implementations and analyses these using two ideas from software testing: property-based testing [9] and mutation testing (see [19] for a recent survey). For property-based testing, our implementation uses Python's Hypothesis [15] library.

3 HEURISTIC AND IMPLEMENTATION

Our heuristic is based on two key ideas:

- If an LLM is given an ambiguous purpose statement for a function and then prompted to generate multiple implementations, it *may* generate two or more *functionally inequivalent* implementations.
- Inputs which demonstrate that two implementations are functionally inequivalent *may* reveal ambiguities in the purpose statement.

Our use of the word *may* reflects our uncertainty that the heuristic we have developed on the basis of these ideas will be effective in practice. We defer this concern to Section 4. For now, we describe our heuristic and its implementation in GuardRails in detail. The input to our heuristic is a function's signature which specifies the type of each argument, an optional purpose statement that may be ambiguous, and zero or more functional examples. We illustrate our heuristic for the example presented in Section 1.1, as shown in Figure 2.

(1) Based on the inputs given, we use an LLM to suggest an initial set of syntactically valid implementations for the function. GuardRails works on suggestions provided by GitHub Copilot, which is available as an extension in Visual Studio Code, a popular IDE. Besides inline suggestions, Copilot also provides an option to view the top 10[8] suggestions in a panel. GuardRails picks up the suggestions from this panel and then uses them in its functionality. When it is invoked, GuardRails retains only syntactically valid implementations in a *suggestion space.* In Figure 2, this suggestion space initially contains five implementations, three of which are shown.

(2) We augment this suggestion space by mutating each implementation. In GuardRails, we use a modified fork of the mutation testing tool MutPy[9] to mutate all the initial implementations. In Figure 2, we add 10 mutants, resulting in a suggestion space of 15 implementations.

(3) We attempt to fuzz [25] each implementation in the suggestion space by executing it on multiple inputs, as per the function's signature. Fuzzing can cause some implementations to fail on certain inputs (e.g., a division or modulus operation may fail when the second operand is `0`, or a list `max()` operation may fail on an empty list). In Figure 2, we discover the input `[]` during this fuzzing step. In GuardRails, we use Hypothesis' Ghostwriter module to automatically generate a strategy for passing inputs to fuzz each implementation[10]. To ensure that Hypothesis generates inputs of the appropriate type, we require the function's signature to contain type hints (also known as type annotations). We record any inputs that cause such an implement to fail. Some implementations fail because they exceed GuardRails's upper-limit on the execution time (currently, 0.3 seconds per test). This occurs because syntactically valid suggestions may contain infinite loops or may implement an algorithm that is inefficient for some inputs generated by Hypothesis.

(4) If some functional examples have been provided, we discard all LLM-generated implementations that fail on any of these examples. For GuardRails, we assume that functional examples have been specified as doctests. We use Python's `doctest` module to discard implementations that fail one or more doctests, thereby trimming the suggestion space. In Figure 2, this trims the suggestion space from 15 implementations to just 5 implementations.

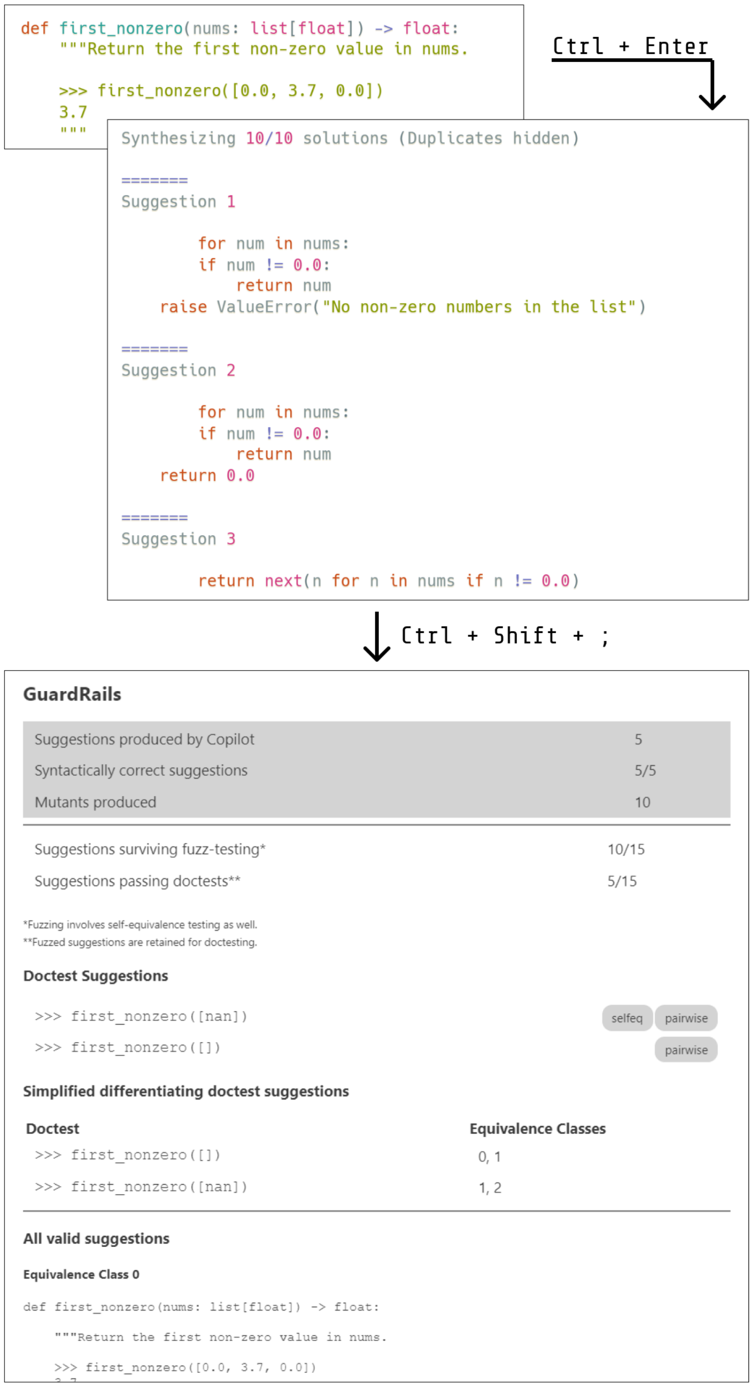
(5) For each pair of implementations in the resulting suggestion space, we attempt to find an input on which these implementations are functionally inequivalent. In GuardRails, we create pair-wise equivalence tests and use the input strategy created in Step (3) to run these tests using Hypothesis, for finding such an input. Hypothesis uses heuristics to *shrink* failing inputs[11], often resulting in simple, easy-to-read tests. In Figure 2, we discover the input `[nan]` during this step.

(6) Finally, we collate all recorded inputs and present these to the programmer. In GuardRails, we report these as *partial* doctests i.e., we specify the input and explicitly prompt the programmer to provide the expected output (Figure 2). This is in contrast to Copilot Chat, which assumes an expected output without revealing the potential ambiguity (Figure 1).

GuardRails is open-sourced and is available as a GitHub repository[12]. It is implemented as a VSCode Extension, a link to which can

[8]This number can be changed in the configurations.

[9]https://github.com/mrigankpawagi/mutpy
[10]https://hypothesis.readthedocs.io/en/latest/ghostwriter.html
[11]https://hypothesis.readthedocs.io/en/latest/data.html#shrinking
[12]https://github.com/mrigankpawagi/GuardRails



```
def first_nonzero(nums: list[float]) -> float:
    """Return the first non-zero value in nums.

    >>> first_nonzero([0.0, 3.7, 0.0])
    3.7
    """
```

```
Synthesizing 10/10 solutions (Duplicates hidden)

=======
Suggestion 1

        for num in nums:
        if num != 0.0:
            return num
    raise ValueError("No non-zero numbers in the list")

=======
Suggestion 2

        for num in nums:
        if num != 0.0:
            return num
    return 0.0

=======
Suggestion 3

        return next(n for n in nums if n != 0.0)
```

GuardRails

Suggestions produced by Copilot	5
Syntactically correct suggestions	5/5
Mutants produced	10
Suggestions surviving fuzz-testing*	10/15
Suggestions passing doctests**	5/15

*Fuzzing involves self-equivalence testing as well.
**Fuzzed suggestions are retained for doctesting.

Doctest Suggestions

```
>>> first_nonzero([nan])
>>> first_nonzero([])
```

Simplified differentiating doctest suggestions

Doctest	Equivalence Classes
`>>> first_nonzero([])`	0, 1
`>>> first_nonzero([nan])`	1, 2

All valid suggestions

Equivalence Class 0

```
def first_nonzero(nums: list[float]) -> float:

    """Return the first non-zero value in nums.

    >>> first_nonzero([0.0, 3.7, 0.0])
```

Figure 2: An illustration of our heuristic and implementation for the `first_nonzero()` function.

be found in the repository along with a link to the performance comparison dataset discussed in Section 4. Once installed in VSCode, GuardRails can be invoked by first triggering Copilot's suggestions panel, and then pressing an appropriate key combination[13].

[13]`Ctrl+Shift+;` runs the full heuristic as described here. `Ctrl+Shift+/` skips step (2) i.e., it does not generate any mutants. For some problems, MutPy generates an excessive number of mutants, resulting in degraded performance.

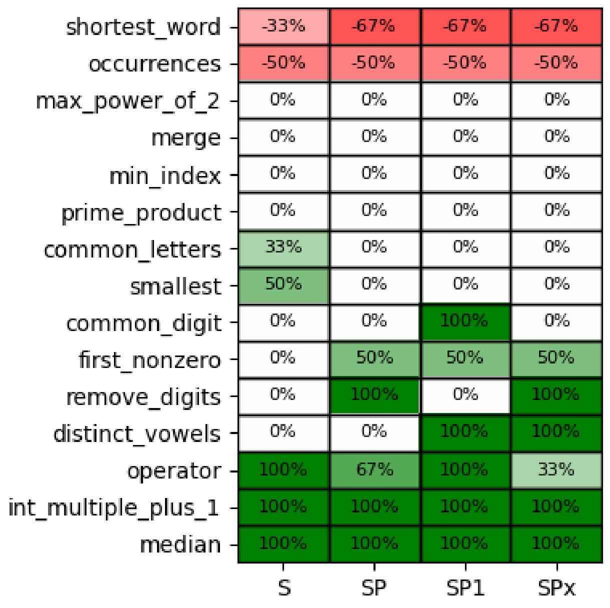


Figure 3: Differences in percentages of AIC (for each variant of all 15-questions) caught by GuardRails and GitHub Copilot Chat (top@5).

4 COMPARISON WITH COPILOT CHAT

To the best of our knowledge, Copilot's Chat feature relies *only* on a suitably trained LLM when prompted to generate unit tests (Figure 1). In contrast, GuardRails prompts the underlying LLM to generate code, and this code is evaluated using additional tools (the doctest and Hypothesis modules, and MutPy). We expect this additional computation to result in an improved ability to identify inputs from AICs. Both tools rely on LLMs, whose results are not deterministic [5]. Hence, for each (function, variant) combination, we executed both tools 5 times and we report the best result (top@5).

4.1 RQ1: Relative Performance

A comparison of the abilities of Copilot Chat relative to GuardRails is shown in Figure 3, for the 15 functions and their 4 variants. The numbers within each row are similar, indicating that differences in relative performance are due to the specifics of each function. Figure 3 shows that Copilot Chat identifies inputs from more AICs than GuardRails for 2/15 functions (red), the performance of the two tools is similar for 7/15 functions (largely white), and GuardRails is better for 6/15 functions (largely green).

4.2 RQ2: Absolute Performance by Variant

When we average the performance across all functions, we find that both tools are able to leverage increasing levels of detail to a similar extent (Figure 4). However, while Copilot Chat is able to raise the average percentage of AICs found from 49% (variant **S**) to 68% (variant **SPx**), GuardRails starts from a higher base of 69% (variant **S**) and improves to 93% (variant **SPx**).

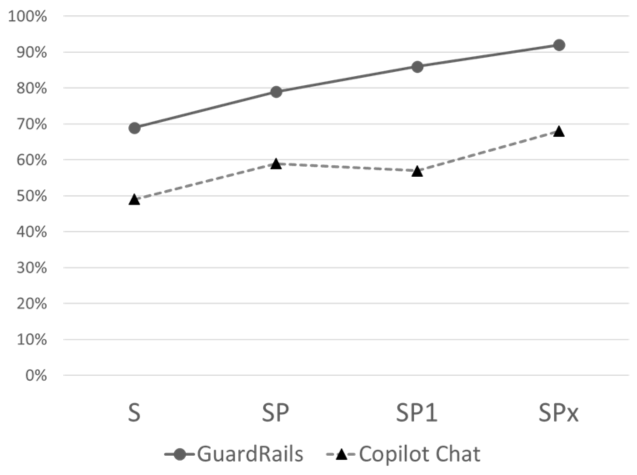


Figure 4: The percentage of AICs (averaged over all 15 questions) found by GitHub Copilot Chat vs. GuardRails (top@5).

5 LIMITATIONS

Although the results of our heuristic are promising, we acknowledge that GuardRails is a research prototype with several limitations. We believe that these limitations can be addressed so that our heuristic can be incorporated into professionally developed tools such as Copilot Chat. Our key limitations are:

(1) We have focused on making GuardRails usable for only one programming language (Python 3) in the specific context of *individual* functions. This prevents our tool from being used for whole programs, or even for functions that call helper functions (e.g., functions imported from other modules). Further, to ensure that Hypothesis can ghostwrite accurate tests, functions must have type hinting.
(2) Since GuardRails relies on the underlying LLM to generate complete implementations, it is presently suitable for simple problems (e.g., CS1 level problems). As LLMs continue to improve, our heuristic could be applicable for more complex problems [13, 17].
(3) Since key components of GuardRails are not deterministic (LLMs [5] and Hypothesis [15]) our tool occasionally produces poor results. A similar limitation applies to Copilot Chat, which is why we report top@5 results in Section 4.

6 DISCUSSION AND FUTURE WORK

6.1 Usage by Instructors

We believe that instructors will find GuardRails useful in two ways. First, while creating code-writing tasks (e.g., for high-stakes examinations), instructors can use our tool to check problem statements for ambiguities. We believe that it would be particularly interesting to investigate the utility of GuardRails for this purpose in a linguistically diverse country such as India. Second, instructors may wish to deliberately write ambiguous problem statements [23]. In this case, GuardRails can be used to confirm that the ambiguities in the specification are the ones expected.

6.2 Usage by Novice Programmers

We have demonstrated principled ways in which even novice programmers can increase the level of detail presented to LLMs (from variant **S** to variant **SPx**) in order to improve their ability to detect ambiguities. As we have acknowledged in Section 5, GuardRails can be used in limited contexts. We hypothesize that exposure to ambiguities highlighted by our tool can help novices develop the ability to identify ambiguities in broader contexts as well. This line of research seems particularly promising for students with limited fluency in the language in which problems are specified.

7 CONCLUSIONS

We have proposed a novel heuristic that uses LLMs to identify potential ambiguities in the purpose statements of Python functions. Further, we compared our open-source tool, GuardRails, against a production-level benchmark (Copilot Chat). We have demonstrated that our tool can *explicitly* identify potential ambiguities and is often (but not always) able to outperform Copilot Chat. We hope that the ideas presented here are incorporated into widely used, professionally developed tools such as GitHub Copilot. We believe our heuristic can further enhance the productivity of software developers and also empower novice programmers. Finally, we believe that our heuristic can support new approaches to CS pedagogy and assessment that expose students to deliberately ambiguous problem specifications.

REFERENCES

[1] 2008. IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2008* (2008), 1–70. https://doi.org/10.1109/IEEESTD.2008.4610935

[2] GS Adithi, Akshay Adiga, K Pavithra, Prajwal P Vasisht, and Viraj Kumar. 2015. Secure, Offline Feedback to Convey Instructor Intent. In *2015 IEEE Seventh International Conference on Technology for Education (T4E)*. IEEE, 105–108.

[3] Rishabh Balse, Bharath Valaboju, Shreya Singhal, Jayakrishnan Madathil Warriem, and Prajish Prasad. 2023. Investigating the Potential of GPT-3 in Providing Feedback for Programming Assessments. In *Proceedings of the 2023 Conference on Innovation and Technology in Computer Science Education V. 1* (Turku, Finland) *(ITiCSE 2023)*. Association for Computing Machinery, New York, NY, USA, 292–298. https://doi.org/10.1145/3587102.3588852

[4] Brett A. Becker, Paul Denny, James Finnie-Ansley, Andrew Luxton-Reilly, James Prather, and Eddie Antonio Santos. 2023. Programming Is Hard - Or at Least It Used to Be: Educational Opportunities and Challenges of AI Code Generation. In *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1* (Toronto ON, Canada) *(SIGCSE 2023)*. Association for Computing Machinery, New York, NY, USA, 500–506. https://doi.org/10.1145/3545945.3569759

[5] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé de Oliveira Pinto, Jared Kaplan, Harrison Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating Large Language Models Trained on Code. *CoRR* abs/2107.03374 (2021). arXiv:2107.03374 https://arxiv.org/abs/2107.03374

[6] Dev Dave, Angelica Celestino, Aparna S. Varde, and Vaibhav Anu. 2022. Management of Implicit Requirements Data in Large SRS Documents: Taxonomy and Techniques. *SIGMOD Rec.* 51, 2 (jul 2022), 18–29. https://doi.org/10.1145/3552490.3552494

[7] Paul Denny, Viraj Kumar, and Nasser Giacaman. 2023. Conversing with Copilot: Exploring prompt engineering for solving CS1 problems using natural language. In *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1*. 1136–1142.

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

[9] George Fink and Matt Bishop. 1997. Property-Based Testing: A New Approach to Testing for Assurance. *SIGSOFT Softw. Eng. Notes* 22, 4 (jul 1997), 74–80. https://doi.org/10.1145/263244.263267

[10] James Finnie-Ansley, Paul Denny, Brett A Becker, Andrew Luxton-Reilly, and James Prather. 2022. The Robots Are Coming: Exploring the Implications of OpenAI Codex on Introductory Programming. In *Australasian Computing Education Conference*. 10–19.

[11] Shuvendu K. Lahiri, Aaditya Naik, Georgios Sakkas, Piali Choudhury, Curtis von Veh, Madanlal Musuvathi, Jeevana Priya Inala, Chenglong Wang, and Jianfeng Gao. 2022. Interactive Code Generation via Test-Driven User-Intent Formalization. arXiv:2208.05950 [cs.SE]

[12] Juho Leinonen, Arto Hellas, Sami Sarsa, Brent Reeves, Paul Denny, James Prather, and Brett A Becker. 2023. Using Large Language Models to Enhance Programming Error Messages. In *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1*. 563–569.

[13] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. 2022. Competition-level code generation with alphacode. *arXiv preprint arXiv:2203.07814* (2022).

[14] Shu Lin, Na Meng, Dennis Kafura, and Wenxin Li. 2021. PDL: Scaffolding Problem Solving in Programming Courses. In *Proceedings of the 26th ACM Conference on Innovation and Technology in Computer Science Education V. 1* (Virtual Event, Germany) *(ITiCSE '21)*. Association for Computing Machinery, New York, NY, USA, 185–191. https://doi.org/10.1145/3430665.3456360

[15] David R MacIver, Zac Hatfield-Dodds, et al. 2019. Hypothesis: A new approach to property-based testing. *Journal of Open Source Software* 4, 43 (2019), 1891.

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

[17] OpenAI. 2023. GPT-4 Technical Report. arXiv:2303.08774 [cs.CL]

[18] Maciej Pankiewicz and Ryan S Baker. 2023. Large Language Models (GPT) for automating feedback on programming assignments. *arXiv preprint arXiv:2307.00150* (2023).

[19] Mike Papadakis, Marinos Kintis, Jie Zhang, Yue Jia, Yves Le Traon, and Mark Harman. 2019. Mutation testing advances: an analysis and survey. In *Advances in Computers*. Vol. 112. Elsevier, 275–378.

[20] James Prather, Raymond Pettit, Brett A. Becker, Paul Denny, Dastyni Loksa, Alani Peters, Zachary Albrecht, and Krista Masci. 2019. First Things First: Providing Metacognitive Scaffolding for Interpreting Problem Prompts. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education* (Minneapolis, MN, USA) *(SIGCSE '19)*. Association for Computing Machinery, New York, NY, USA, 531–537. https://doi.org/10.1145/3287324.3287374

[21] Arun Raman and Viraj Kumar. 2022. Programming Pedagogy and Assessment in the Era of AI/ML: A Position Paper. In *Proceedings of the 15th Annual ACM India Compute Conference* (Jaipur, India) *(COMPUTE '22)*. Association for Computing Machinery, New York, NY, USA, 29–34. https://doi.org/10.1145/3561833.3561843

[22] Sami Sarsa, Paul Denny, Arto Hellas, and Juho Leinonen. 2022. 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*. 27–43.

[23] G. Michael Schneider. 1978. The Introductory Programming Course in Computer Science: Ten Principles. In *Papers of the SIGCSE/CSA Technical Symposium on Computer Science Education* (Detroit, Michigan) *(SIGCSE '78)*. Association for Computing Machinery, New York, NY, USA, 107–114. https://doi.org/10.1145/990555.990598

[24] Leonardo Silva, António José Mendes, Anabela Gomes, and Gabriel Fortes. 2023. Investigating Programming Students Problem Comprehension Ability and its Association With Learning Performance. *IEEE Transactions on Education* 66, 2 (2023), 156–162. https://doi.org/10.1109/TE.2022.3204906

[25] Ari Takanen, Jared D Demott, Charles Miller, and Atte Kettunen. 2018. *Fuzzing for software security testing and quality assurance*. Artech House.

[26] Michel Wermelinger. 2023. Using GitHub Copilot to Solve Simple Programming Problems. (2023).

[27] John Wrenn and Shriram Krishnamurthi. 2019. Executable Examples for Programming Problem Comprehension. In *Proceedings of the 2019 ACM Conference on International Computing Education Research (ICER '19)*. 131–139. https://doi.org/10.1145/3291279.3339416

[28] John Wrenn and Shriram Krishnamurthi. 2020. Will Students Write Tests Early Without Coercion?. In *Proceedings of the 20th Koli Calling International Conference on Computing Education Research* (Koli, Finland) *(Koli Calling '20)*. Association for Computing Machinery, New York, NY, USA, Article 27, 5 pages. https://doi.org/10.1145/3428029.3428060