Jiang, B. et al. (Eds.) (2025). Proceedings of the 33rd International Conference on Computers in Education. Asia-Pacific Society for Computers in Education

Ambiguity Resolution with Human Feedback for Code Writing Tasks

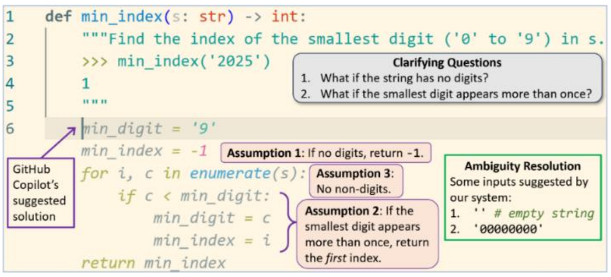
Aditey NANDAN[a] **& Viraj KUMAR**[b*]
[A]*Mathematics and Computing, Indian Institute of Science, India*
[b]*Kotak-IISc AI-ML Centre, Indian Institute of Science, India*
*viraj@iisc.ac.in

Abstract: Specifications for code writing tasks are usually expressed in natural language and may be ambiguous. Programmers must therefore develop the ability to recognize ambiguities in task specifications and resolve them by asking clarifying questions. We present and evaluate a prototype system, based on a novel technique (ARHF: Ambiguity Resolution with Human Feedback), that (1) suggests specific inputs on which a given task specification may be ambiguous, (2) seeks limited human feedback about the code's desired behavior on those inputs, and (3) uses this feedback to generate code that resolves these ambiguities. We evaluate the efficacy of our prototype, and we discuss the implications of such assistive systems on Computer Science education.

Keywords: Computer Science education, Ambiguous Specifications, Code LLMs.

1. Introduction

Real-world specifications for code writing tasks are often expressed in natural language, and such specifications can be ambiguous (Jinwala & Shah, 2015). A failure to resolve such ambiguities early in the software development lifecycle can lead to costly errors (Fernández et al., 2017). Therefore, existing computing curricula such as CS2023 (ACM/IEEE-CS/AAAI Joint Task Force, 2023) emphasize the importance of requirements elicitation (pg. 242) and regard the ability of students to assess the "lack of ambiguity" in technical documents as a core learning outcome (pg. 280). Since students have easy access to powerful Generative AI tools, and since these tools are increasingly being used in professional software development contexts, Becker et al. (2023) have highlighted an urgent need to review educational practices in computing. In this spirit, we ask the following question: Can assistive systems help programmers identify and resolve ambiguities in task specifications? If so, we are interested in understanding the impact of such systems on computing education. In this paper, we limit our attention to code writing in the context of introductory programming courses (CS1).



```
1   def min_index(s: str) -> int:
2       """Find the index of the smallest digit ('0' to '9') in s.
3       >>> min_index('2025')
4       1
5       """
6       min_digit = '9'
        min_index = -1
        for i, c in enumerate(s):
            if c < min_digit:
                min_digit = c
                min_index = i
        return min_index
```

Clarifying Questions
1. What if the string has no digits?
2. What if the smallest digit appears more than once?

Assumption 1: If no digits, return -1.

Assumption 3: No non-digits.

Assumption 2: If the smallest digit appears more than once, return the *first* index.

Ambiguity Resolution
Some inputs suggested by our system:
1. `'' # empty string`
2. `'00000000'`

Figure 1. A solution with assumptions generated by GitHub Copilot (Line 6 onwards) for an ambiguous code-writing task (problem P1 in this study, Lines 2 to 5). A good programmer would ask clarifying questions instead of making assumptions to resolve ambiguities.

Figure 1 shows a typical CS1 code-writing task with a specification that is ambiguous because it is incomplete: the purpose statement (or docstring) on Line 2 and the example (or doctest) on Lines 3 and 4 fail to specify the code's expected behavior on certain inputs. As Schneider (1978) notes, such ambiguous specifications can be interpreted in multiple "reasonable" ways that are functionally inequivalent. Before writing any code, a good programmer should ask clarifying questions to distinguish between these inequivalent interpretations. In contrast, modern code-generation tools such as GitHub Copilot often write code by making assumptions. For instance, the code in Figure 1 makes two seemingly "reasonable" assumptions: the function should return `-1` if the input string `s` has no digits (Assumption 1), and if the smallest digit occurs multiple times in `s`, it should return the index of the *first* occurrence (Assumption 2). However, we think it is "unreasonable" for the code to assume that all letters in `s` are digits (Assumption 3). We explore whether assistive tools can help programmers recognize that a given task supports multiple reasonable interpretations, and can help distinguish between these interpretations by suggesting clarifying questions. If so, we hope to identify skills that students will need to develop if such tools become widely available. We propose a technique, Ambiguity Resolution with Human Feedback (ARHF), whose novelty stems from the insight that code generation models can be prompted to (1) use an ambiguous task specification to generate *specific* inputs on which the code's desired behavior *may* be ambiguous, and (2) use human feedback that clarifies the code's desired behavior on these specific inputs to produce code that handles *general* inputs correctly. We have developed a prototype system that uses ARHF, and we have evaluated it on six CS1-level code-writing tasks in Python. Our research questions are:

- **RQ1**: How effectively can our ARHF system generate inputs that distinguish between all reasonable interpretations of an ambiguous task specification?
- **RQ2**: How effectively can our ARHF system use human feedback to generate accurate code given an ambiguous task specification?

2. Related Work

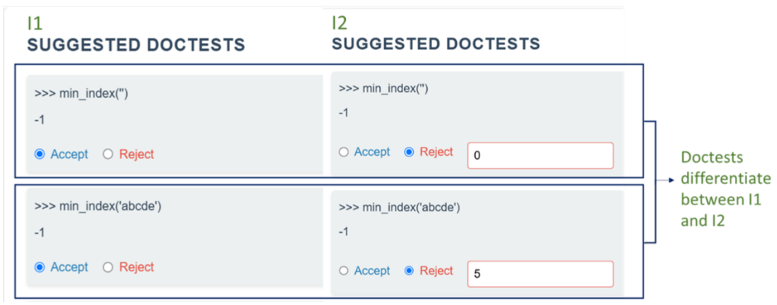
As powerful Generative AI tools are being integrated into professional workflows, computing education researchers are rethinking pedagogy, assessments, and learning outcomes (Prather et al., 2023, Raman & Kumar, 2022). For instance, code-writing tasks are typically simple and well-specified, and large language models trained to generate code (henceforth: Code LLMs) can solve such tasks reliably (Prather et al., 2023). Thus, recent research has explored Probeable Problems (Pawagi & Kumar, 2024; Denny et al., 2025): code-writing tasks where the specification can be interpreted in multiple ways because it lacks certain details. To solve such problems, programmers must issue 'probes' to clarify the target interpretation before writing code. In contrast, our approach has numerous similarities to interactive test-driven code generation (Lahiri et al., 2022). Although the task specification *S* may be ambiguous (i.e., it may be consistent with multiple "reasonable" interpretations), both approaches assume that the programmer has a target implementation *X* in mind. Both use a Code LLM to generate a candidate implementation *C* from specification *S* and a set of test inputs *T* that may reveal differences between multiple interpretations of *S*. Both execute implementation *C* on each input in *T*, and both rely on feedback from the programmer to determine whether *C* matches the target implementation *X*. The key difference is the nature of the human feedback: whereas Lahiri et al. (2022) seek "lightweight" Boolean feedback (i.e., whether or not implementation *C* has the desired behavior on each input in *T*), we expect programmers to accurately specify the behavior of *X* whenever this differs from the behavior of *C* on an input in *T*. Hebbar et al. (2025) call this skill reactive task comprehension.

3. Methods

3.1 ARHF System Architecture

Our ARHF system is implemented as a web based, locally run application. The frontend is built using HTML, CSS and JS, while the backend is implemented using Python (Flask). The input to ARHF is a (potentially ambiguous) specification *S*. For this study, we assume that *S* is a Python function *signature* (the function name as well as the name and type-hint for each argument), a *docstring* explaining the function's purpose, and at least one *doctest* (an example showing the desired output as per the target interpretation on some test input). ARHF has 3 components: Initial Code Generation (which generates a candidate implementation *C* from *S*), Test Input Generation (which generates test inputs *T* from *S* and *C*), and Code Correction (which attempts to fix implementation *C* using human feedback on *C*'s output for each input in *T*). All components use Code LLMs. Our prototype system uses a common Code LLM: Qwen2.5-Coder-32B-Instruct (Hui et al., 2024) with temperature = 0.7, top_p = 0.8, and top_k = 20, via the Hugging Face Inference API (https://huggingface.co/Qwen/Qwen2.5-32B-Instruct/blob/main/generation_config.json). The detailed prompts for each component are at: https://github.com/Nandan-Aditey/ARHF-Resources/tree/main/ARHF-Prompts

In addition to using a Code LLM, the Test Input Generation component generates test inputs targeting edge cases using CrossHair (Schanely, 2017). Each input in this combined set *T* is then executed on the implementation *C*, and our system displays each input-output pair as a doctest. As an example, Figure 2 shows two such doctests for two interpretations of Problem P1 (`min_index`). The implementation *C* is equivalent to interpretation I_1: the function returns `-1` when string `s` contains no digit. If I_1 is the desired interpretation, the programmer can "Accept" both doctests (as shown in Figure 2 left). If there are no failing doctests, our system proposes *C* as the desired implementation. On the other hand, suppose the desired interpretation is I_2: the function returns `len(s)` when string `s` contains no digit. In this case, the programmer can "Reject" the doctests and specify the correct outputs: `0` for the empty input string, and `5` for the string `'abcde'` (Figure 2, right). Given these failing doctests and implementation *C*, the Code Correction component must inductively deduce that the correct return expression when string `s` has no digits is `len(s)`. If the revised implementation again fails some doctests, it is possible to rerun the Code Correction component. However, we do not perform reruns in this study. Instead, our system presents the "best" of the two generated implementations (the function that passes the maximum number of doctests, and the most recently generated function in case of a tie).


I1
SUGGESTED DOCTESTS

```
>>> min_index('')
-1
```
[✓ Accept] Reject

```
>>> min_index('abcde')
-1
```
[✓ Accept] Reject

I2
SUGGESTED DOCTESTS

```
>>> min_index('')
-1
```
Accept [✓ Reject] 0

```
>>> min_index('abcde')
-1
```
Accept [✓ Reject] 5

Figure 2. ARHF's interface to obtain human feedback. The two doctests differentiate between interpretations I_1 and I_2 for Problem P1 (`min_index`).

3.2 Quantitative Analysis

We evaluate our system using two types of problems. First, we created three problems with multiple interpretations (P1, P2, P3 in Table 1). Treating each of these interpretations as the target, we attempt to use ARHF to generate the desired implementation (25 times for each interpretation). In addition, we selected three Probeable Problems proposed by Denny et al. (2025), each with 16 "reasonable" interpretations (P7, P8, P9 in Table 1). We lightly modified problem P8 to return a value rather than to print an answer. For these problems, we fixed the target interpretation as the one chosen by Denny et al. (2025), and we attempted to use ARHF to generate the corresponding implementation (100 times each).

Table 1. *The six problems in this study, each with multiple "reasonable" interpretations. The implementations corresponding to each of these interpretations are available at:* https://github.com/Nandan-Aditey/ARHF-Resources/tree/main/ARHF-Problems

Problem	Docstring	Initial Doctest	Interp.
P1	Find the index of the smallest digit ('0' to '9') in *s*.	>>> min_index('2025') 1	4
P2	Find the minimum frequency. Return None if *data* is empty.	>>> min_freq([1, 2, 1]) 2	3
P3	Count the number of digits in *n*.	>>> num_digits(123) 3	2
P7	Count the number of integers in data between a and b.	>>> count_between([1, 2, 3], 0, 5) 3	16
P8	Search data for the smallest even value.	>>> smallest_even([50, 25, 2, 30, 45]) [2]	16
P9	Find the first vowel in s.	>>> first_vowel('apple') 'a'	16

For each problem and each target interpretation *X*, we evaluate the set of test inputs *T* generated by the Test Input Generation component of our system. Specifically, for each non-target interpretation *Y*, we check whether there is at least one input in *T* on which *Y* and *X* disagree. If so, we say that the set of inputs *T* distinguishes interpretation *X* from *Y*. To answer RQ1, we define Input Ambiguity Resolution (IAR) as the fraction of non-target interpretations that are distinguished by the inputs in *T*. Further, we check whether the implementation *C* presented by ARHF is functionally equivalent to *X*. We check for equivalence by manual inspection, and we use the Pass@1 metric (Chen et al., 2021) to estimate the probability that *C* is functionally equivalent to *X*. Finally, we define Code Ambiguity Resolution (CAR) as the fraction of non-target interpretations that are functionally inequivalent to *C*. Since equivalence with *X* implies inequivalence with every non-target interpretation, the Pass@1 metric is always upper bounded by CAR. We use both metrics to answer RQ2.

4. Results

The following table summarizes the performance of ARHF on the 6 problems in this study.

Table 2. *IAR, CAR, and Pass@1 of our ARHF system for the six problems in this study*

Prob.	IAR	CAR	Pass@1	Remarks
P1	(3×25)/(3×25) = 1.0	(4×75)/(4×3×25) = 1.0	(4×25)/(4×25) = 1.0	Each interp. 25 times
P2	(2×25)/(2×25) = 1.0	(50+48+47)/(3×2×25) = 0.97	(25+23+22)/(3×25) = 0.93	
P3	(1×25)/(1×25) = 1.0	(25+0)/(2×1×25) = 0.5	(25+0)/(2×25) = 0.5	
P7	1495/(15×100) = 0.99	1489/(15×100) = 0.99	95/100 = 0.95	Single interp. 100 times
P8	1473/(15×100) = 0.98	1323/(15x100) = 0.88	61/100 = 0.61	
P9	1412/(15×100) = 0.94	1370/(15x100) = 0.91	0/100 = 0.0	

4.1 RQ1: How effectively do ARHF-generated test inputs resolve ambiguities?

To answer RQ1, we evaluate the IAR of our system for the six problems in this study. Our results (Table 2) demonstrate that ARHF achieves the highest possible IAR (1.0) for problems with only a few reasonable interpretations (P1, P2, and P3) and it maintains a high IAR (at least 0.94) even when problems admit greater ambiguity (P7, P8, and P9). Our system's performance is poorest on problem P9. To understand why, we analyze the test inputs and code generated by ARHF in relation to the 16 "reasonable" interpretations for each problem. When the target interpretation is I_7, ARHF correctly distinguishes the non-target interpretation I_8 from I_7 in only 12% of invocations. Interpretation I_7 is: "first vowel in vowel order, ignoring differences in upper-case and lower-case vowels". The test inputs generated by ARHF are often unable to distinguish this from interpretation I_8: "first vowel in index order, ignoring differences in upper-case and lower-case vowels". Since the task specification lacks so many crucial details, it is perhaps unsurprising that ARHF cannot generate appropriate test inputs. Nevertheless, the high IAR score indicates that the inputs generated can help eliminate many reasonable interpretations from further consideration. Thus, ARHF-generated test inputs appear to be reasonably effective at resolving ambiguities.

4.2 RQ2: How effectively does ARHF resolve ambiguities in the generated code?

To answer RQ2, we evaluate both the Pass@1 and CAR rates of ARHF for each of the six ambiguous code-writing problems. ARHF generally achieves high Pass@1 and CAR rates, except for problems P3 and P9.

For P3, the Pass@1 rate is 0 when the target interpretation is I_2: "count the unique number of digits". Recall that our system achieves an IAR of 1.0 for P3 regardless of which interpretation was chosen as the target (Table 2). Thus, when the target implementation is I_2, the doctests created from human feedback on test inputs generated by our system are inconsistent with interpretation I_1. Nevertheless, the Code LLM in our system generates an implementation corresponding to interpretation I_1 of P3 on all 25 invocations. This suggests an inability of current Code LLMs to generate code that strictly adheres to provided doctests. This limitation of Code LLMs has been noted previously by Heo et al. (2024).

The particularly poor performance on problem P9 is unsurprising since the test inputs are insufficiently rich to compensate for the many missing details in the task specification (see Section 4.1). For P9, 95% of ARHF invocations reveal an implementation corresponding to interpretation I_8 instead of the target interpretation I_7. While the Pass@1 rate is zero, the high CAR rate (0.91) shows that ARHF successfully rejected most of the unwanted interpretations. While this is promising, we also note that when the implementation presented by ARHF is *inequivalent* to the target implementation, it corresponds to an interpretation that is slightly different from the one desired. In other words, when the implementation generated by ARHF is incorrect, it is likely to be subtly buggy!

5. Limitations, Future Work, and Conclusion

We note two key limitations of this study and discuss how they can be addressed in future work. First, our study is limited to just six code-writing tasks (specifically, writing a single function), in a single programming language (Python). Future work can explore a richer variety of code-writing tasks. Second, although our results demonstrate that ARHF can be effective at resolving ambiguities, our system requires the programmer to accurately specify the desired outputs on inputs suggested by our system. It will therefore be useful to evaluate whether students, especially novice programmers, can make effective use of a system such as ours. We are presently evaluating ARHF in a CS1 course setting. We reiterate that our system is a prototype that can be improved in several ways, such as by rerunning the Code Generation module when doctests fail. It is also possible that breakthroughs in Code LLMs will improve their ability to adhere to specified doctests. Thus, the results reported in this paper should be viewed as a conservative lower bound on the performance of such systems.

In conclusion, this paper has presented a prototype system that uses off-the-shelf components to effectively resolve ambiguities in code-writing task specifications using a novel strategy: Ambiguity Resolution with Human Feedback (ARHF). We have shown that although ARHF is imperfect when key details are missing in the specification, it is nevertheless helpful in rejecting unwanted interpretations of that specification. Since such systems can reduce the risk of costly errors in software development (Fernández et al., 2017), we anticipate that systems more capable than our prototype will become readily available. For educators, such systems could be helpful in two ways. First, code-writing tasks in summative assessments often need to be clear, and systems such as ours could help detect potentially ambiguous wording. Second, learning outcomes and assessments associated with identifying ambiguities (Pawagi and Kumar, 2024; Denny et al., 2025) may require rethinking if students have access to powerful assistive ARHF-like systems.

Acknowledgements

The authors respectively acknowledge support from the Intel AI PC Experiment Development Centre (FA/OTHR-24-0031) and the Kotak-IISc AI-ML Centre towards this research.

References

ACM/IEEE-CS/AAAI Joint Task Force. (2023). Computer Science Curricula 2023 (CS2023): The Final Report. ACM Press, IEEE Computer Society Press and AAAI Press.

Becker, B. A., Denny, P., Finnie-Ansley, J., Luxton-Reilly, A., Prather, J., & Santos, E. A. (2023, March). Programming is hard-or at least it used to be: Educational opportunities and challenges of ai code generation. In *Proc. of the 54th ACM Technical Symposium on CS Education V. 1* (pp. 500-506).

Chen, M., Tworek, J., Jun, H., Yuan, Q., Pinto, H. P. D. O., Kaplan, J., ... & Zaremba, W. (2021). Evaluating large language models trained on code. arXiv preprint arXiv:2107.03374.

Denny, P., Kumar, V., MacNeil, S., Prather, J., & Leinonen, J. (2025). Probing the Unknown: Exploring Student Interactions with Probeable Problems at Scale in Introductory Programming. In *Proceedings of the 30th ACM Conference on Innovation and Technology in Computer Science Education* (pp. 618-624). DOI: 10.1145/3724363.3729093.

Fernández, D. M., Wagner, S., Kalinowski, M., Felderer, M., Mafra, P., Vetrò, A., ... & Wieringa, R. (2017). Naming the pain in requirements engineering: Contemporary problems, causes, and effects in practice. *Empirical software engineering*, 22, 2298-2338.

Hebbar, S. V., Harini, S., & Kumar, V. (2025). Refuting LLM-generated Code with Reactive Task Comprehension. In *Proceedings of the 30th ACM Conference on Innovation and Technology in Computer Science Education* (pp. 30-36). DOI: 10.1145/3724363.3729100

Heo, J., Heinze-Deml, C., Elachqar, O., Chan, K. H. R., Ren, S., Nallasamy, U., Miller, A., & Narain, J. (2025). Do LLMs “know” internally when they follow instructions? In Proceedings of the International Conference on Learning Representations (ICLR 2025). *arXiv:2410.14516* [cs.AI].

Hui, B., Yang, J., Cui, Z., Yang, J., Liu, D., Zhang, L., ... & Lin, J. (2024). Qwen2. 5-coder technical report. *arXiv:2409.12186*.

Jinwala, D. C., & Shah, U. S. (2015). Resolving Ambiguities in *Natural Language Software Requirements*. ACM SIGSOFT Software Engineering Notes, 40, 1-7.

Lahiri, S. K., Fakhoury, S., Naik, A., Sakkas, G., Chakraborty, S., Musuvathi, M., ... & Gao, J. (2022). Interactive code generation via test-driven user-intent formalization. *arXiv:2208.05950*.

Pawagi, M., & Kumar, V. (2024). Probeable Problems for Beginner-level Programming-with-AI Contests. In *Proceedings of the 2024 ACM Conference on International Computing Education Research-Volume 1* (pp. 166-176).

Prather, J., Denny, P., Leinonen, J., Becker, B. A., Albluwi, I., Craig, M., ... & Savelka, J. (2023). The robots are here: Navigating the Generative AI Revolution in Computing Education. In *Proceedings of the 2023 Working Group reports on Innovation and Technology in CS Education* (pp. 108-159).

Raman, A., & Kumar, V. (2022, November). Programming pedagogy and assessment in the era of AI/ML: A position paper. In *Proceedings of the 15th ACM India Compute Conference* (pp. 29-34).

Schanely, P. (2017). CrossHair - An analysis tool for Python that blurs the line between testing and type systems. https://github.com/pschanely/CrossHair.

Schneider, G. M. (1978, February). The introductory programming course in computer science: ten principles. In *Papers of the SIGCSE/CSA technical symposium on CS education* (pp. 107-114).