Ambiguity by Design: Practicing Requirement Clarification through Natural-Language Dialogue with LLMs

Kay Tang
University of Auckland
Auckland, New Zealand
ktan185@aucklanduni.ac.nz

Hoanh Nguyen Hosea Tong-Ho
University of Auckland
Auckland, New Zealand
hton892@aucklanduni.ac.nz

Kaitlin Riegel
University of Auckland
Auckland, New Zealand
kaitlin.riegel@auckland.ac.nz

Paul Denny
University of Auckland
Auckland, New Zealand
paul@cs.auckland.ac.nz

Nasser Giacaman
University of Auckland
Auckland, New Zealand
n.giacaman@auckland.ac.nz

Juho Leinonen
Aalto University
Espoo, Finland
juho.2.leinonen@aalto.fi

Abstract

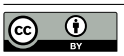
A widely used pedagogy in introductory programming courses involves students solving small code-writing exercises with well-defined problem statements. Although such exercises help students practice basic programming skills and become familiar with syntax, they offer little opportunity to interpret or clarify ambiguous requirements. However, real-world programming rarely provides such clarity. Developers must interpret incomplete specifications, ask questions, and resolve inconsistencies before writing code. At the same time, advances in large language models (LLMs) have made well-specified programming problems trivial to solve, allowing students to obtain correct solutions with little effort when task requirements are explicit. In this paper, we present a web-based tool that delivers 'Probeable Problems' which are programming tasks with deliberately ambiguous specifications that require students to clarify requirements before coding. Our tool extends prior implementations by enabling students to engage in a natural language dialogue with an AI "client" to uncover and resolve ambiguities, alongside a traditional mechanism for probing behaviour using code inputs. We deployed the tool in a large introductory programming course and examined how students engaged with the conversational interface. Students reported that the tool helped them appreciate the importance of asking specific questions and valued the realism of interacting with a simulated client, even when they found the tasks challenging. We also found that greater exploration of ambiguities was associated with fewer failed attempts, indicating that deliberate inquiry before coding supports more effective problem solving.

CCS Concepts

• **Social and professional topics → Computing education**.

Keywords

Probeable Problems, CS1, programming education, large language models, test cases, requirements, ambiguity



ACE 2026, Melbourne, VIC, Australia

ACM ISBN 979-8-4007-2352-0/26/02
https://doi.org/10.1145/3786228.3786249

ACM Reference Format:
Kay Tang, Hoanh Nguyen Hosea Tong-Ho, Kaitlin Riegel, Paul Denny, Nasser Giacaman, and Juho Leinonen. 2026. Ambiguity by Design: Practicing Requirement Clarification through Natural-Language Dialogue with LLMs. In *28th Australasian Computing Education Conference (ACE 2026), February 09–13, 2026, Melbourne, VIC, Australia.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3786228.3786249

1 Introduction

Introductory programming courses have historically used coding exercises with well-defined problem descriptions to both teach and assess student learning. However, this approach can limit opportunities to interpret vague problem statements or elicit missing requirements which are critical skills in professional software development [13]. Failure to clarify requirements early can lead to misaligned code that fails to meet client expectations, causing expensive delays and rework [13]. Consequently, current CS curricula emphasise pre-coding competencies such as client communication and requirements gathering to provide immediately applicable skills [19].

Moreover, the emergence of large language models (LLMs) capable of generating correct code from natural language prompts has reduced the usefulness of traditional coding exercises. LLMs can solve most CS1 and CS2 problems from course assignments and exams [14, 15, 31], allowing students to obtain correct solutions with minimal effort when specifications are unambiguous. This undermines the intended learning benefits of such tasks and raises academic integrity concerns. Both the need to teach requirements clarification and the increasing redundancy of traditional programming problems has motivated efforts for alternative pedagogical approaches.

One relatively new approach is the idea of 'Probeable Problems' introduced by Pawagi and Kumar [28]. Probeable Problems include an intentionally vague problem statement and a mechanism that allows students to *probe* the program's expected behaviour [9, 28]. In prior implementations, these probes were submitted in the form of code inputs to a model solution (or 'oracle') which would then generate and display the expected output.

In this paper, we build on prior work by presenting a novel implementation of Probeable Problems where students interact with an LLM-powered oracle (a 'client') using natural language. Students ask clarifying questions, and when those questions are

sufficiently precise, the client reveals relevant details, enabling exploration of the problem space. Natural language probing may be beneficial as it mirrors authentic client communication and can alleviate the burden from students of having to translate lines of questioning into program inputs.

To evaluate our tool and to compare how students explore ambiguities using natural language compared to more traditional input probing (i.e., code fragments), we investigate the following research questions:

RQ1: How does student exploration of the problem space differ when probing using natural language compared to code input probes?

RQ2: What are students' experiences with the tool and the client?

RQ3: How does student exploration of the problem space correlate with their programming-task performance?

2 Related Work

2.1 Ambiguity in Programming Education

Calls to expose students to ambiguous problem statements date back decades [33], yet most introductory programming exercises tend to present fully-specified descriptions [1]. While these enable syntax practice and the development of basic programming skills, they leave little opportunity to practice elicitation skills or discuss specifications. Recent curriculum guidance (CS2023) emphasises competencies beyond programming, such as communication and requirements clarification [19], which are skills highly valued in professional software development [13]. Furthermore, LLMs have trivialised well-defined problems by solving them swiftly and accurately [14, 15]. Beyond academic integrity concerns [31], students may also (rightly or wrongly) begin to question the value of tasks that can be solved automatically.

2.2 Utilising LLMs in Programming Pedagogy

When leveraged thoughtfully, LLMs enable novel pedagogical approaches that teach and tutor students [2, 22]. Becker et al. argue that with AI-generated code now embedded in education, the emphasis should shift from code writing to code reading and evaluation, to prepare students for industry [3]. An emerging complementary skill is formulating effective prompts for LLMs. Recent approaches such as 'prompt problems' give students opportunities to practise this skill by describing programming tasks to an AI model rather than writing code directly [10, 32]. This is one example of a broader pedagogical trend in which effective interaction with AI systems is becoming a core component of programming literacy.

Beyond prompting, parallel work leverages LLMs to scaffold learning rather than provide direct solutions. Tools like the CodeAid platform emphasise conceptual explanations, pseudo-code with reasoning, and targeted annotations of code over direct answer-giving [20]. More broadly, recent classroom evidence identifies the desirable characteristics students want from AI teaching assistants: instant access around peak times, engaging support that preserves learner autonomy, and guardrails that guide problem solving step-by-step instead of revealing full solutions [11]. There is clear potential for LLMs to act as tutors that scaffold and guide student reasoning, an approach that aligns closely with our goal of supporting students as they clarify ambiguous programming requirements.

2.3 Probeable Problems

Probeable Problems, introduced by Pawagi and Kumar [28], are problems with intentionally vague problem statements, along with an 'oracle', which is a mechanism for seeking clarifications regarding expected behaviour. Initially introduced in a beginner-level programming contest that allowed the use of AI coding tools, Probeable Problems proved resistant to modern, easy-to-access AI code generation tools [28]. Denny et al. explored the idea in a more traditional learning context, finding that systematic probing by students was correlated with higher success and fewer incorrect submissions [9]. In addition, students reportedly found the problems challenging but authentic and reflective of real-world clarification tasks.

In both of these prior studies, Probeable Problems were framed to students as "Ask the Client" questions. However, constructing these *questions* required students to build syntactically correct code fragments. These code fragments, that essentially set up input variables, could be submitted as probes to the 'oracle' which would generate the expected output for the given inputs. Through an iterative process of *probing* the oracle students could deduce the intended behaviour of the program (and then implement it).

3 Platform Implementation

In the current work, we present a web-based tool designed for delivering Probeable Problems, which incorporates an interface for students to talk with a simulated client in natural language to clarify ambiguities. Alternatively, students may write code inputs, and probe the client to see the expected program behaviour (similar to previous implementations reported in the literature). Problems are presented in a two-stage process: (1) a requirements clarification stage, and (2) an implementation stage where, once the requirements are clear, code to solve the problem can be implemented.

3.1 Functionality

3.1.1 Requirements Clarification. Figure 1 shows the user interface for the requirements clarification stage in our tool. The top half of the figure shows the natural language interface, where students can enter messages in a simulated chat with the client. This pane is displayed by selecting "Client" from the menu bar. The bottom half of the figure shows the code input probing mechanism ('oracle'), where students can set up a code input probe by initialising variables and then call the function which they ultimately need to implement. This pane is displayed by selecting "Run" from the menu bar.

3.1.2 Natural Language Design Considerations. Prior to building the natural language interface, we conducted initial prototyping and user testing to gauge the feasibility of an LLM-powered client. In our first iteration of the system prompt, we provided the LLM (GPT-4o, Temperature: 0.5, TopP: 0.7) with the model solution and task instructions. We tested our prototype with a rudimentary Command Line Interface (CLI), sending questions through the CLI and printing out the LLMs response in a while loop. We observed that the LLM had a tendency to (1) overshare about what the function should do, often describing the implementation in full, and (2) respond

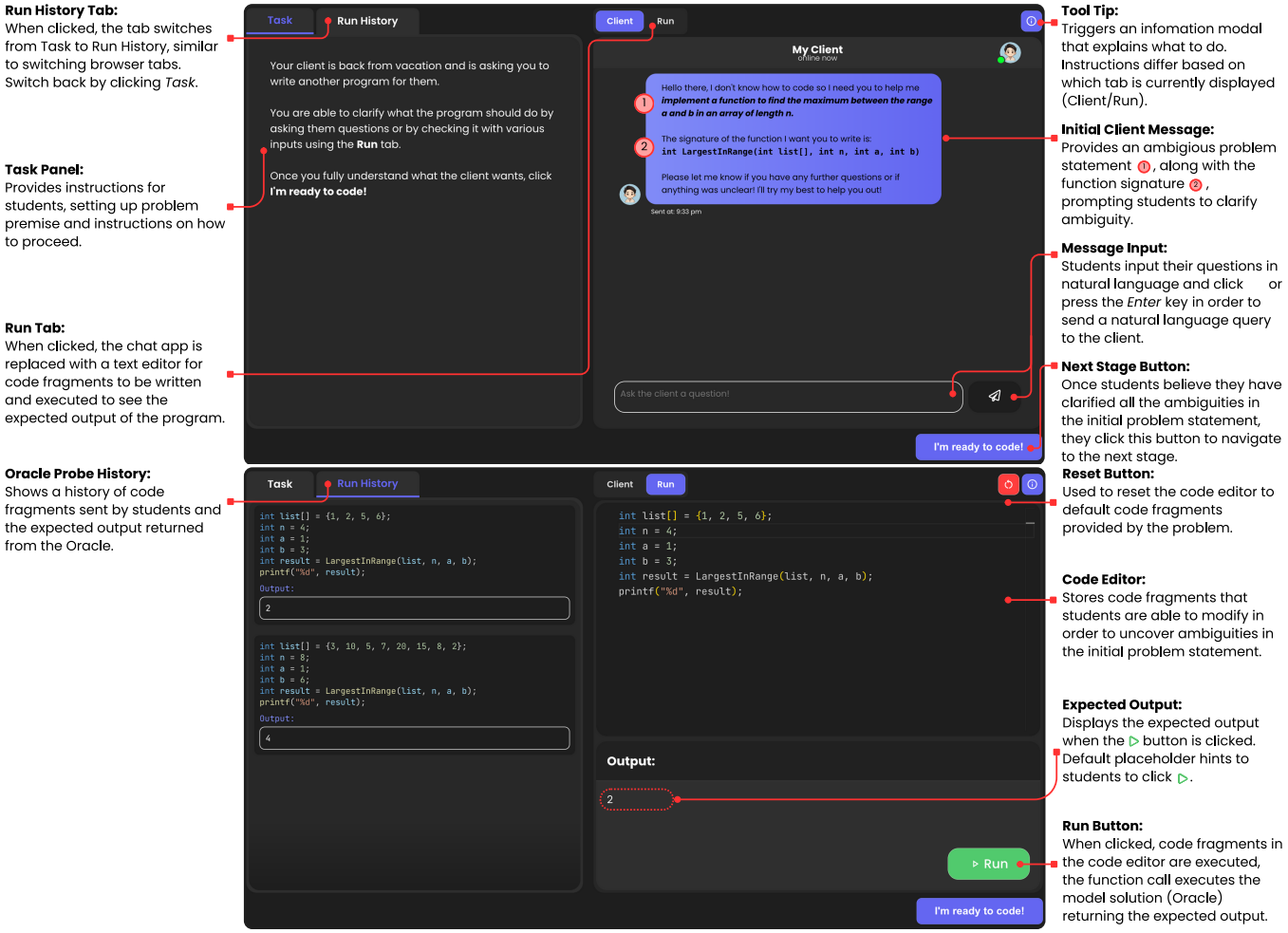


Figure 1: Requirements clarification stage for the problem *Implement a function to find the maximum between the range a and b in an array of length n*, showing the two different methods of probing explored in our study. Top: natural language probes, bottom: code input probes.

with erroneous function behaviour. These incorrect responses were consistent with known LLM hallucination risks [18]. From this initial prototyping, we formulated three concrete requirements for the client: (i) it must not reveal more than what is asked of it by the student, (ii) it should encourage students to ask specific rather than broad questions, and (iii) it must not provide incorrect information about program behaviour.

Our final design of the natural language interface is modelled after modern messaging applications, simulating a text-based dialogue with a person. The student is presented the problem statement and function signature in the initial message from the client. This is annotated in Figure 1.

3.1.3 Client Design. We adapt recent work by Wang et al. [34], controlling the client by a Finite State Machine (FSM)-like system prompt that enforces a strict order of operations and mutually exclusive paths. By using this style of system prompt, we were able to specify paths for how the model should reply to certain types of questions asked by a student. For each question asked, the model returns exactly one JSON object conforming to a predefined schema. Specific behaviour that our client should display, such as rejecting vague questions and encouraging more focused clarification, deflecting attempts to obtain the model solution, and refusing code reading/writing without exception, are achieved through guardrails that prevent the model from deviating from its original instructions.

The client answers questions specifically targeting certain ambiguities. Consider the problem, *"Implement a function to find the maximum between the range a and b in an array of length n."* Asking *"What should the function do?"* typically generates a response like *"Can you be more specific about what you would like to know?"*, as the question is vague. In contrast, an accepted question would be *"Could you explain to me what a and b are?"* where an example response would be *"a: a boundary index for the range. b: a boundary index for the range. Let me know if you have any more questions!"*, since the question seeks to uncover ambiguity regarding two input

Figure 2: Constraints for the problem: *Implement a function to find the maximum between the range a and b in an array of length n.* These constraints are embedded in the system prompt for the LLM to answer QT2 natural language probes.

1. If n is less than 0 then the function must return -1.
2. If either boundary (a or b) is less than 0 the function must return -1.
3. If either boundary (a or b) is greater than or equal to n the function must return -1.
4. The order of a and b can be swapped and the function will produce the same result.
5. The "range" is exclusive of the elements a and b.
6. The function should print the smallest even value's index, not the actual even value.
7. If the distance between a and b is less than 1 the return -1.

parameters. In this case, the client deliberately does not reveal if the range is inclusive or exclusive.

Conceptually, the system prompt is designed to make the model behave like a deterministic program: each type of student question triggers a specific, predefined response path, ensuring consistent and constrained outputs. This is aligned with the "prompting-is-programming" perspective [4]. We use OpenAI's o4-mini model as it is trained to *"think before answering"* with deliberate, step-by-step internal reasoning. This enables the model to follow our rigid FSM-like system prompt more reliably than standard chat models like GPT-4o [24–27]. We chose o4-mini over other OpenAI reasoning models due to its low latency and lower cost per token.

3.1.4 Ensuring Correctness of Outputs. To ensure output quality, we use a mechanism that utilises the system prompt and model solution. In the system prompt, the LLM distinguishes between two possible categories of questions that students can ask when clarifying ambiguities. The first question-type (**QT1**) includes questions that aim to uncover the expected behaviour given a specific set of inputs. For example, a question in this category would be: *"When* `values` *contains 1, 3, 5 and* `length` *is 3, what should the output be?"* The second question-type (**QT2**) includes questions that aim to uncover a specific ambiguity in the initial problem statement. For example, *"Is the range inclusive of a and b?"*. Note that, although the LLM differentiates and handles these types of questions differently, both question types can be used to uncover the same ambiguity.

To ensure responses to **QT1**s do not contain erroneous information, we use the LLM to extract from the student's natural language message the code input required to test expected behaviour of the program (essentially, a code input probe) and we execute this against the model solution for the problem to generate an objectively correct answer. We then format the generated output and return the result in a templated response. This process is illustrated in Figure 3. The tool observes for compilation or runtime errors during program execution in the rare event that malformed test inputs are returned by the LLM. In these situations the client returns a templated response prompting the student to ask the question again later as they are currently "busy".

For **QT2**s, the LLM is provided with the model solution along with a set of constraints that the function must adhere to. These constraints are effectively the ambiguities that need to be found by the student. An example of the constraints provided to the model is shown in Figure 2. The model is instructed to check this list of constraints before responding to the question, identify the best matching constraint, and use it when formatting a response.

3.1.5 Solution Generation. In the implementation stage of the problem, we provide students with a code editor and the ability to submit their code for checking, where it is executed against a comprehensive test suite. When there are test case failures, the tool enables the viewing of all test cases passed until the first failure. Subsequent test cases are intentionally locked to discourage students from discovering remaining ambiguities from viewing test cases. When a student receives a compilation error only the first error message is displayed, following guidance from prior work on simplifying error message reporting to students [12].

We also provide the option for students to write a natural language description of the problem which they can submit to a simulated 'AI agent' which will generate code based on their description. This allows students to practice writing clear, unambigious problem specifications, in a similar fashion to prior work on prompt problems [10]. The 'AI agent' we designed, called 'Cogs', is powered by OpenAI's GPT-4o model. This underlying model has been shown to perform well on rigorous code-generation benchmarks [21], and is thus suitable for powering our agent.

4 Methods

4.1 Study Background

We explored our natural language extension of Probeable Problems in the context of a large introductory programming course in C (N = 996), taught at the University of Auckland in the second half of 2025 (approved by the University of Auckland Human Participants Ethics Committee #25279). The course is compulsory for all first-year engineering students. We reused an existing problem from Denny et al. [9] (**P1**) and developed two new Probeable Problems (**P2** and **P3**). Table 1 lists the problem statements and summarises their ambiguities.

The tool was deployed in a single week-long lab alongside other programming exercises. The lab contributed 1% to the course grade and students were able to complete it remotely or on campus. To receive the credit, students were required to complete a tutorial question and questions P1-3. No penalties were given for incorrect submissions. Instead, to encourage students to engage with exploring the problem space in stage one (opposed to iteratively updating their solution after failing a test case), we gamified the tool by implementing a reward system using in-tool points. Students were able to compare how many points they had earned against their peers on a public leaderboard built in the tool.

Students attempted the problems in a fixed order: tutorial, P1, P2, P3. P1's interface was restricted to the client interface (natural language probes) and P2 to the oracle interface (code input probes). For P3, students had both the client interface and the oracle interface available to them to use.

In order to evaluate student experiences of the natural language extension to Probeable Problems (**RQ2**), students responded to a brief survey, which included two Likert items ranging from *Strongly*

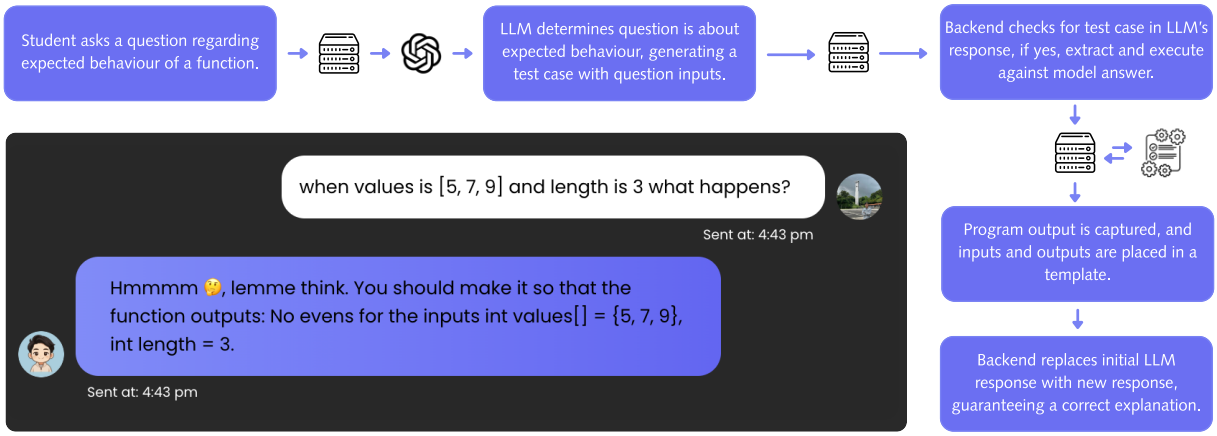


Figure 3: Flow diagram showing how the tool ensures the client provides students with the correct information regarding the expected behaviour of the function.

Table 1: Probeable Problem statements, a brief summary of the ambiguities that students must identify to solve the problem, and the problem variant determining the probing interfaces provided for that problem.

Problem	Statement	Summary of ambiguities	Variant
P1	Implement a function to search an array of length n for the smallest even value	*multiple indices may qualify, print in descending order, special error if no evens*	client
P2	Implement a function that finds a trend in an array of length n	*print special values if array is strictly increasing or decreasing, special error for invalid length*	oracle
P3	Implement a function to find the maximum between the range a and b in an array of length n	*maximum refers to index, 'a' and 'b' can be specified in either order, and they are strictly excluded from the range*	both

Disagree (1) to *Strongly Agree* (5) and one open-response question delivered after engaging with the problems:

- **Item 1:** *I preferred using the "Client" tab (i.e. asking questions) over the "Run" tab (i.e. executing tests) when finding out information about the task requirements.*
- **Item 2**: *Probeable Problems helped me understand the importance of asking highly specific questions when clarifying task requirements.*
- **Open-response:** *Describe your overall experience using Probeable Problems.*

4.2 Classification of Probes

To analyse the depth of student exploration of the problem space through their probing, we first defined how to classify a probe. We will discuss this using the context of P1.

Consider the probe $P1_0$: `int values = [1, 3, 5], int length = 3`, which aims to uncover the ambiguity: *what should happen if there are no evens?* Now consider a different probe $P1_1$: `int values = [7, 1, 9, 7], int length = 4`. Both probes target the same ambiguity, as both the oracle and the client will reveal the expected behaviour is to output "No evens". This means that it is possible for a student to probe numerous times with varying inputs, but not uncover any new ambiguities.

We define an Ambiguity Class as the set of probes that uncover the same ambiguity. In the following sections, we will refer to a specific Ambiguity Class as $AC_{\text{<ambiguity>}}$.

4.2.1 Classification of Code Input Probes. To classify code input probes submitted to the oracle, we created a set of alternative implementations for a problem, where each version intentionally contained a single, isolated ambiguity. When a student submitted a probe, we executed it against both the correct solution and all of these variant implementations. If a variant produced a different output than the correct solution, we inferred that the probe had triggered that specific ambiguity and assigned it to the corresponding Ambiguity Class.

The classification process for code input probes is illustrated in Figure 4. For example, this highlights a buggy implementation for P1 when the array contains no even value. This buggy implementation prints "NO EVENS", which is not the expected "No evens", and thus we would classify the probe into $AC_{\text{No evens}}$.

4.2.2 Classification of Natural Language Probes. Classification of natural language probes involves having to classify both **QT1** and **QT2**. In the former scenario, since we run the code input extracted by the LLM from the natural language message against the model solution, we use the same method as classifying code input probes. For **QT2**, since these questions do not contain specific inputs, we

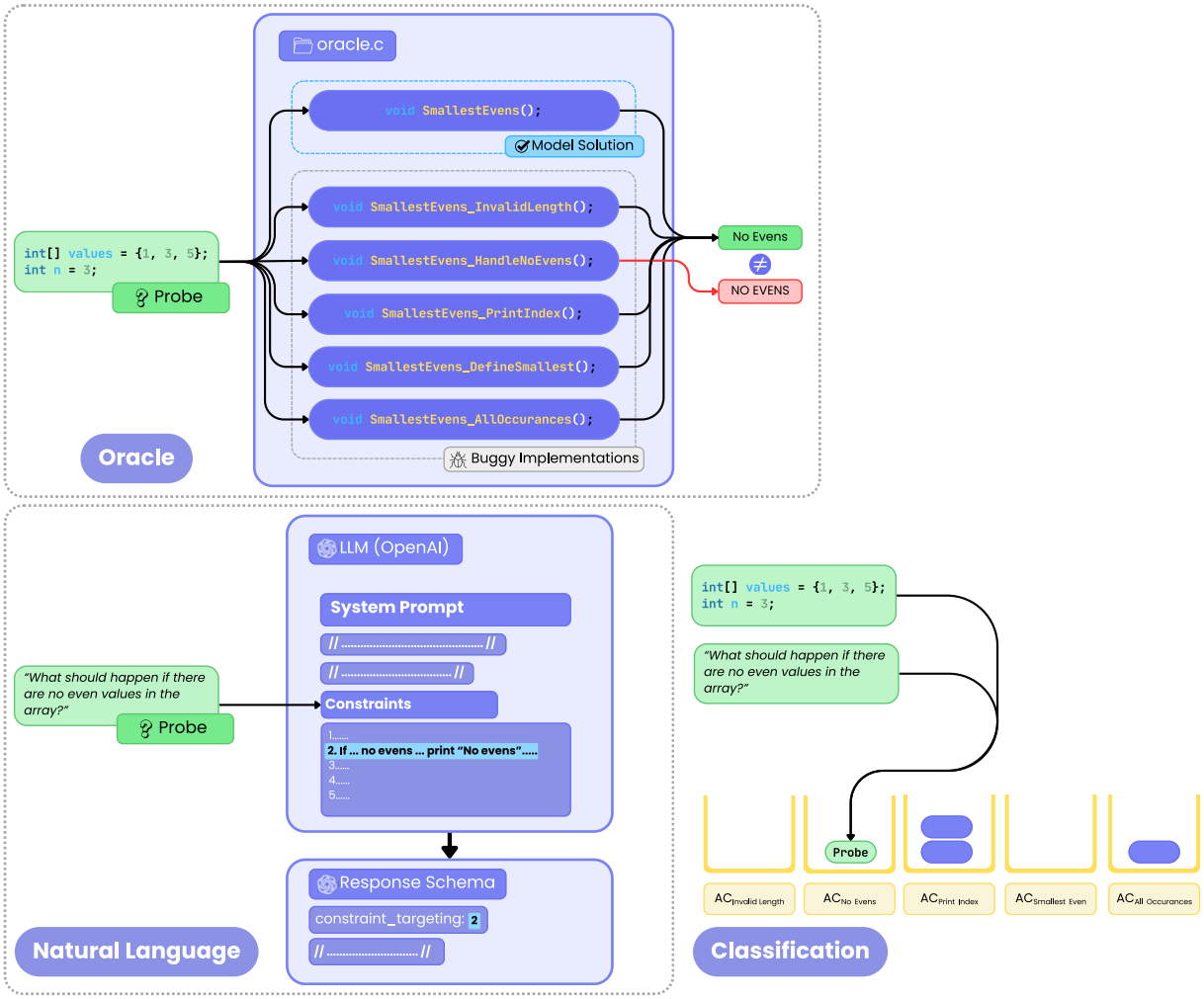


Figure 4: Classification of probes into their respective *ACs* for P1. Code input probes and natural language QT1 probes are classified through the outputs of buggy implementations and the model solution (top) while natural language QT2 probes are classified by the LLM (bottom left). Bottom right shows how a probe is classified into $AC_{\text{No evens}}$ in code and natural language.

Table 2: Results of LLM (OpenAI's o4-mini) classification of QT2 questions into their respective Ambiguity Classes.

Problem	Correct	Incorrect	Accuracy
P1	70	5	93.3%
P2	73	2	97.3%
P3	105	0	100%
Total	248	7	97.3%

rely on the LLM (o4-mini) to classify the probe into the correct *AC*. Figure 4 illustrates this classification process, which involves the LLM checking the embedded list of constraints and selecting the constraint which is most relevant to the question.

To measure the reliability of the LLM classification, we systematically evaluated its ability to classify a variety of **QT2** questions. For each ambiguity in P1-3, we wrote five potential example questions that students could ask that the model should classify into its respective *AC*. We called the model through the API, including the system prompt, initial client message, and potential question in the context window. We then processed the response from the model, checking that the LLM's classification matched the constraint the question is supposed to target. We repeated this process three times for every question. For P1-3, the number of constraints given to the model were five, five, and seven, respectively. As shown in Table 2 we observed a 97.3% accuracy in the model's ability to classify **QT2**, which we find sufficient for our analysis.

4.3 Data Analysis

994 students engaged with the tool, either on the tutorial problem or on one of the three questions that are analysed in this study. We first wished to compare how thoroughly students explored the problem space when probing using natural language compared to code input

probes (**RQ1**). To do this, we use the anonymised data from a prior study by Denny et al. [9] (collected in the same course one year prior), in which students could only write code input probes. In our current study, students attempted the equivalent question (**P1**), but did so using natural language probes to the client only. Thus, we could compare the thoroughness of the student probes in each case, comparing code input and natural language probes for the same problem.

Toward understanding students' experiences of the tool (**RQ2**), we first analysed the responses to the survey items. 954 students responded to at least one question on the survey. Using the Likert items, we conducted one-sample *t*-tests to detect significant deviations from a *neutral* response (argued as appropriate in [23]). An experienced researcher oversaw a codebook inductive thematic analysis [6] on the open-ended question. As student responses were short, they were coded with a single code that captured the most prominent idea being conveyed. A subset of responses was initially coded by a researcher. Codes were then jointly reviewed by the first two authors and organised into preliminary themes. Due to the volume of responses, 200 randomly ordered responses were coded by the first two authors. New codes were discussed and revised between the researchers. Coding continued until no new codes emerged (~250), indicating thematic saturation [16]. Themes were then revised and finalised. Using **P3**, where students were free to choose whether to submit natural language or code input probes, we sought to determine if there was a significant difference between how they opted to probe. As the data violated the assumption for a *t*-test, we opted to use a Sign test.

Finally, to test how exploration of the problem space related to student success (**RQ3**), we examined the association between the number of *ACs* students discovered and their failed attempts on P1–3. We used the definition of *attempt* from Denny et al. [9], and consider an attempt *successful* once all test cases within the test suite of a problem are passed. Once a student achieves a successful attempt, we stop recording probes and any new failing code submissions. We counted the number of failed attempts for each student until they were successful. Across P1–3, fewer than 1% of attempts were successful without any probes. We observed these attempts involved pasting a complete solution into the code editor that passed all test cases without further modification. We are confident these were not honest attempts and thus these were excluded from our analysis.

5 Results

5.1 RQ1: Natural Language vs. Code Inputs

A break down of the type of probes and attempts submitted across the three problems is included in Table 3. Investigating the difference between students' exploration of the problem space using natural language probes compared to code input probes, we specifically looked at P1.

We performed post-processing on the 16,402 probes collected for P1 in Denny et al. [9] (P8 in their paper) through the buggy implementations, classifying each probe into their respective *AC*. In the prior study, a total of 965 attempts were recorded for P1, while in our study, 922 attempts were recorded. We observed the distribution of *ACs* discovered in the prior study to be left-skewed

Table 3: Overview of problem attempts across P1–P3, showing number of probes broken down by probe type. Natural language (NL) is broken down into QT1 & QT2 probes. CI = code input.

Problem	P1	P2	P3
# Attempts	922	874	831
# Failed Attempts	5613	6986	5394
# Messages sent	5956	–	5754
# NL (QT1) Probes	839	–	1231
# NL (QT2) Probes	3040	–	2735
# CI Probes	–	7765	4875
# Probes Total			20185

Table 4: Descriptive statistics of the Likert items and *t*-test results demonstrating variation from *neutral*. Item 1 *n* = *953* and Item 2 *n* = 954.

	Mean (SD)	Skew (SE)	*t*	*p*	95%CI	*d*
Item 1	2.88 (1.13)	-0.06 (0.08)	-3.28	0.001	-0.19, -0.05	0.11
Item 2	3.74 (1.13)	0.92 (0.08)	20.16	<.001	0.67, 0.81	0.65

(M = 3.46, SD = 1.02), while in our study, the distribution appears more normally distributed (M = 2.07, SD = 1.32).

Using Welch's *t*-test, the cohort in our study discovered significantly fewer constraints on average than the prior cohort ($t(1736)$ = -25.63, p < .001), with a very large effect (95% CI [-1.50, -1.29], d = -1.19). On further investigation of this result, we found the average number of probes students made in our study (M = 4.20, SD = 2.99), was significantly fewer than than the average number of probes made in Denny et al. [9] (M = 17.00, SD = 15.17) (Welch's $t(1042)$ = -25.71, p < 0.001).

5.2 RQ2: Student Perspectives

In this section, we outline the results of the survey responses and their observable behaviours to unpack students' perspectives on the tool.

5.2.1 Likert Responses. Table 4 presents the descriptive statistics and significant changes from a *neutral* response to each Likert item. We found that students were less likely to prefer the *Client* tab (natural language probes) to the *Run* tab (code input probes) ($t(952)$ = -3.28, p = .001), but not by a large margin (M = -0.12, d = 0.11). Students agreed that the tool helped them understand the importance of asking highly specific questions when clarifying task requirements ($t(953)$ = 20.16, p < .001), with a reasonable effect size (M = 0.74, d = 0.65).

5.2.2 Thematic Analysis. Our thematic analysis resulted in five key themes: *Difficulty and Frustration*, *Novelty and Enjoyment*, *Real-World Applicability*, *Specificity in Elicitation*, and *Assumption Testing*.

Difficulty and Frustration. A dominant theme (n = 143) was the difficulty, and often corresponding frustration, of Probeable Problems. This was discussed generally, but was sometimes linked to specific causes, such as the ambiguous nature of the client's

responses, a preference for more traditional coding approaches, the time required, and problems using the interface. One student commented,

> *It was very frustrating, as it felt like I was failing test by test due to the lack of questions asked. There was always some specific case that didn't work, and I kept having to update the code because of it.*

Students cited the client being "*vague*" or that it would "*leave out really important pieces of information unless you asked them specifically.*" Together, this would suggest that the program, while working as intended, is not necessarily positively received.

Novelty and Enjoyment. In contrast, a number of students (n = 44) indicated that they found the platform novel or enjoyable. Again, this was discussed generally, but students sometimes gave particular reasons, including enjoying the process. For example, one student commented, "*It was a very unique experience, and I enjoyed coding and asking questions to figure out what to actually code*".

However, these responses were often qualified by students noting they also found the tasks difficult or frustrating. One student wrote, "*...while it was a bit frustrating at times, it's still very fun and rewarding when you create a solution that meets all the requirements and constraints.*"

Specificity in Elicitation. Another common theme (n = 39) was that students noticed, learned, and valued the need to narrow down their questions. They sometimes expanded on how it was necessary to ask questions that were highly meticulous in order to receive appropriate answers. One student wrote, "*I actually found it a good learning experience. I was learning to check as many edge cases that I could think of as well as asking the client very specific questions about what they actually wanted.*"

Unsurprisingly, this was one of the greatest barriers to solving the problems. Some students found it challenging to ask precise questions. One commented, "*I struggled a little bit with finding the right questions to ask, and making them specific enough to have all my question answered...*" Although this was perceived as a difficult component of the tasks, it is notable that students reported identifying the value of specificity and learning from the process.

Real-World Applicability. Several students (n = 12) raised the relevance of having to talk to a client prior to coding, highlighting an appreciation for what professional software development with a client could look like. One student wrote,

> *This task made me realise how, in reality, things are more complicated. To me, the majority of the time is spent on understanding what the client wants and what kind of output they want in different cases. It's really hard to consider everything.*

Assumption Testing. Relatedly, several students (n = 12) touched on the importance of fully understanding the client's needs, testing their initial assumptions of the problem statement. One explained,

> *..There would be times where I'd feel like I knew what the client was asking for, and had all the information needed. However, when I did the code and ran it, I found out that I'd be missing things... Little things that may not have been thought of come up when my tests are not passed, forcing me to realise how crucially important it is for clarifying task requirements.*

Table 5: Pearson's correlations of number of failed attempts before success on P1–3 with the number of *AC*s discovered. $^{*}p < .01$; $^{**}p < .001$. See Table 3 for each n.

	Failed attempts before success		
	P1	**P2**	**P3**
Discovered *AC*s	-.15**	-0.10*	-.15**

5.2.3 Observable Usage Preferences. To analyse interface preference, we examined P3 where students could use both code input and natural language probes. We compared the number of messages students sent to the client with the number of code input probes students sent to the oracle, including those who did not complete a successful attempt (n = 836). Activity was higher for the client than for the oracle (5,766 and 4,892 actions, respectively). Overall, 506 students had more interactions with the client, compared with 237 students who had more code input probe interactions (ties = 93). Although both modes could be used, around 300 students made virtually all of their interactions for P3 (>95% of submitted probes) via the natural language client. A two-sided sign test indicates students used the chat function significantly more compared to code input probes ($p < .001$). This differs from the sentiment expressed by students in the surveys, which suggested they had no preference.

5.3 RQ3: Ambiguity Discovery & Success

To test whether a more thorough exploration of the problem space translates into fewer failed attempts, we examined the association between the number of *AC*s students discovered and their failed attempts on P1–3. Table 5 demonstrates that, across all three problems, more *AC*s uncovered were associated with fewer failed attempts. In all cases the relationship was statistically significant but small [8]. Taken together, the pattern is consistent: students who discover more ambiguities tend to make fewer failed attempts.

6 Discussion

This study examined incorporating a natural language interface into Probeable Problems. We first investigated whether this improved students' exploration of the problem-space when compared to code input probing (**RQ1**). We then examined students' perspectives and experiences with the tool (**RQ2**). Finally, we demonstrated how problem space exploration related to success on the programming task (**RQ3**).

6.1 Comparing Natural Language and Code Input Probing (RQ1)

Superficially, our results indicate that students explored the problem space more thoroughly when using code input probes in the prior study by Denny et al. [9], compared to natural language probes in our study. However, results demonstrated there was a much greater number of probes submitted for the question (**P1**), on average, in

the prior study (16402 vs 3879). This highlights the strong possibility that students were more incentivised to probe in the earlier study due to failed code attempts incurring a small grading penalty. Previous research has shown that efforts and performance drop under assessments with low stakes [35], and grade-based penalties could not be included in our tool for the current study. Fewer probes likely corresponds to fewer *AC*s uncovered (supported by the skewed 2024 data), however we are unable to draw a conclusion about whether, in equivalent circumstances, natural language probing can be more effective in exploring the problem space. Thus, future work is required.

Even with these limitations, our results highlight a practical point: regardless of how people interact with the problem, if the environment does not motivate inquiry, it risks surfacing shallow requirements and could potentially cost teams time reworking the problem [5]. This is similar to authentic practice, where teams now use AI to help draft specifications and tickets quickly; however, this should be paired with a high level of effort in eliciting requirements to ensure a good understanding of the entire problem scope.

6.2 Students' Conflicting Perspectives and Experiences (RQ2)

It is clear from the results that the tool is (for the most part) working as intended, that is, encouraging students to ask highly specific questions in exploring a problem space. This is evident in both the Likert data and through analysis of the open responses. Importantly, students reported recognising that their initial assumptions were incomplete or flawed, so went back to the client to ask further questions to test those assumptions. This suggests that, through Probeable Problems, students are experiencing metacognitive reflection [30] by evaluating their understanding. We find further support from students identifying the value and relevance in engaging with these tasks, aligning with the current direction of CS curricula [19]. However, this did not transfer to how students' actually felt about completing the activities.

The thematic analysis indicated students found the process difficult and frustrating. Positive responses identifying the tool's value were often marred by the same issues. A plausible explanation for such feelings could be due to students' low perceived control over task success [29]. Accordingly, scaffolding students in the process of specifying effective questions may be a useful approach during future iterations.

Notably, despite students' reported perspectives, we found evidence of greater use of the client and natural language probes when compared to the oracle and code input probes. This could be because the client interface is displayed first [7], or potentially due to students finding the client easier and more intuitive to interact with, even if the process itself is frustrating. Another possibility is that students simply experienced uncovering ambiguities as frustrating, rather than use of the client specifically.

6.3 Student Success and Exploration of the Problem Space (RQ3)

While we found that more probing corresponds to significantly fewer failed attempts, the correlation is weak. A plausible explanation is incentive misalignment, as students did not incur real penalties and may have lacked sufficient motivation to clarify requirements before coding, instead clarifying only after failure of a specific test case. As such, we recommend a probe-before-code scaffold that, (i) gates coding until a sufficient number of ambiguities have been surfaced and, (ii) shifts grading to reward the coverage and quality of pre-coding probes rather than penalising submission failures. This recommendation aligns with evidence on test-first practices [17]. However, care is needed in its implementation, as students may be tempted to probe minimally to start coding.

7 Limitations

Gathering requirements was not a skill taught in the course prior to students' interactions with the tool, and requires time to develop. Thus some students struggled with coming up with highly specific questions that were accepted by the client. This could have resulted in frustration for some, who may have changed their approach to completing the problem by brute-forcing submissions.

Another limitation was the restricted responses that the client was allowed to give due to guard-railing in the system prompt. The prompt was designed to mirror the oracle's code input probe functionality without giving additional advantage, but this sometimes restricted the natural flow of conversation. Future versions could relax these constraints to more realistically simulate "Ask the Client"-style interactions. Additionally, some students' reported the client being slow. Switching to a faster model could improve usability, but this must be balanced against potential losses in the quality of step-by-step reasoning, which is a key strength of the current implementation of the client.

We also acknowledge some methodological limitations in this work. For efficiency, only one code (and theme) was assigned to each response during qualitative data analysis, limiting the depth of the thematic analysis. In addition, the comparison study for **RQ1** was not a true randomised controlled trial; differences in incentives likely influenced probing behaviour. Future research should investigate the tool and the different probing mechanisms under equivalent conditions.

Finally, LLM-based classification used in **RQ1** and **RQ2** was accurate about 95% of the time, meaning the findings should be viewed as indicative rather than definitive. The tool currently supports only single functions in C, and all data were collected from one cohort at a single institution, which constrains generalisability.

8 Conclusion

In this paper, we introduced an implementation of Probeable Problems that integrates a natural language interface to simulate real-world client interactions using generative AI. Through dialogue, students can clarify ambiguities in a problem statement and iteratively refine their understanding before implementation. Our findings reinforce prior work showing that while students often find ambiguity frustrating, engaging with it supports metacognitive reflection and fosters skills related to requirements elicitation. Students who explored the problem space more thoroughly also achieved success with fewer code submissions, showing the value of deliberate inquiry before coding. We conclude that Probeable

Problems, coupled with natural language interaction, offer a promising way to develop students' skills in identifying and reasoning about gaps in problem specifications.

Acknowledgments

This work was supported by the Research Council of Finland grant #356114.

References

[1] Joe Michael Allen, Frank Vahid, Alex Edgcomb, Kelly Downey, and Kris Miller. 2019. An Analysis of Using Many Small Programs in CS1. 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, 585–591. doi:10.1145/3287324.3287466

[2] Anishka, Atharva Mehta, Nipun Gupta, Aarav Balachandran, Dhruv Kumar, and Pankaj Jalote. 2024. Can ChatGPT Play the Role of a Teaching Assistant in an Introductory Programming Course? arXiv:2312.07343 [cs.HC] https://arxiv.org/abs/2312.07343

[3] Brett A. Becker, Paul Denny, James Finnie-Ansley, Andrew Luxton-Reilly, James Prather, and Eddie Antonio Santos. 2022. Programming Is Hard – Or at Least It Used to Be: Educational Opportunities And Challenges of AI Code Generation. arXiv:2212.01020 [cs.HC] https://arxiv.org/abs/2212.01020

[4] Luca Beurer-Kellner, Marc Fischer, and Martin Vechev. 2023. Prompting Is Programming: A Query Language for Large Language Models. *Proc. ACM Program. Lang.* 7, PLDI, Article 186 (June 2023), 24 pages. doi:10.1145/3591300

[5] Barry W. Boehm and Victor R. Basili. 2001. Software Defect Reduction Top 10 List. *Computer* 34, 1 (Jan. 2001), 135–137. doi:10.1109/2.962984

[6] Virginia Braun and Victoria Clarke. 2012. *Thematic analysis.* 57–71.

[7] Dana R. Carney and Mahzarin R. Banaji. 2012. First Is Best. *PLoS ONE* 7, 6 (June 2012), e35088. doi:10.1371/journal.pone.0035088

[8] Jacob Cohen. 2013. *Statistical power analysis for the behavioral sciences.* routledge.

[9] Paul Denny, Viraj Kumar, Stephen MacNeil, James Prather, and Juho Leinonen. 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 V. 1 (ITiCSE 2025)*. ACM, 618–624. doi:10.1145/3724363.3729093

[10] Paul Denny, Juho Leinonen, James Prather, Andrew Luxton-Reilly, Thezyrie Amarouche, Brett A. Becker, and Brent N. Reeves. 2024. Prompt Problems: A New Programming Exercise for the Generative AI Era. In *Proceedings of the 55th ACM Technical Symposium on Computer Science Education V. 1* (Portland, OR, USA) *(SIGCSE 2024)*. Association for Computing Machinery, New York, NY, USA, 296–302. doi:10.1145/3626252.3630909

[11] Paul Denny, Stephen MacNeil, Jaromir Savelka, Leo Porter, and Andrew Luxton-Reilly. 2024. Desirable Characteristics for AI Teaching Assistants in Programming Education. In *Proceedings of the 2024 on Innovation and Technology in Computer Science Education V. 1* (Milan, Italy) *(ITiCSE 2024)*. Association for Computing Machinery, New York, NY, USA, 408–414. doi:10.1145/3649217.3653574

[12] Paul Denny, James Prather, and Brett A. Becker. 2020. Error Message Readability and Novice Debugging Performance. In *Proceedings of the 2020 ACM Conference on Innovation and Technology in Computer Science Education* (Trondheim, Norway) *(ITiCSE '20)*. Association for Computing Machinery, New York, NY, USA, 480–486. doi:10.1145/3341525.3387384

[13] D. Méndez Fernández, S. Wagner, M. Kalinowski, M. Felderer, P. Mafra, A. Vetrò, T. Conte, M. T. Christiansson, D. Greer, C. Lassenius, T. Männistö, M. Nayabi, M. Oivo, B. Penzenstadler, D. Pfahl, R. Prikladnicki, G. Ruhe, A. Schekelmann, S. Sen, R. Spinola, A. Tuzcu, J. L. De La Vara, and R. Wieringa. 2017. Naming the pain in requirements engineering. *Empirical Softw. Engg.* 22, 5 (Oct. 2017), 2298–2338. doi:10.1007/s10664-016-9451-7

[14] 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 *Proceedings of the 24th Australasian Computing Education Conference* (Virtual Event, Australia) *(ACE '22)*. Association for Computing Machinery, New York, NY, USA, 10–19. doi:10.1145/3511861.3511863

[15] James Finnie-Ansley, Paul Denny, Brett A. Becker, Andrew Luxton-Reilly, and James Prather. 2023. My AI Wants to Know if This Will Be on the Exam: Testing OpenAI's Codex on CS2 Programming Exercises. In *Proceedings of the Australasian Computing Education Conference (ACE '23)*. ACM. doi:10.1145/3576123.3576134

[16] Greg Guest, Arwen Bunce, and Laura Johnson. 2006. How Many Interviews Are Enough?: An Experiment with Data Saturation and Variability. *Field Methods* 18, 1 (2006), 59–82. arXiv:https://doi.org/10.1177/1525822X05279903 doi:10.1177/1525822X05279903

[17] Liang Huang and Mike Holcombe. 2009. Empirical investigation towards the effectiveness of Test First programming. *Inf. Softw. Technol.* 51, 1 (Jan. 2009), 182–194. doi:10.1016/j.infsof.2008.03.007

[18] Lei Huang, Weijiang Yu, Weitao Ma, Weihong Zhong, Zhangyin Feng, Haotian Wang, Qianglong Chen, Weihua Peng, Xiaocheng Feng, Bing Qin, and Ting Liu. 2025. A Survey on Hallucination in Large Language Models: Principles, Taxonomy, Challenges, and Open Questions. *ACM Transactions on Information Systems* 43, 2 (Jan. 2025), 1–55. doi:10.1145/3703155

[19] Joint Task Force on Computing Curricula (ACM, IEEE-CS, and AAAI). 2023. *Computer Science Curricula 2023: The Final Report.* Technical Report. ACM, IEEE Computer Society, and AAAI. https://ieeecs-media.computer.org/media/education/reports/CS2023.pdf Accessed 26 Sept 2025.

[20] Majeed Kazemitabaar, Runlong Ye, Xiaoning Wang, Austin Zachary Henley, Paul Denny, Michelle Craig, and Tovi Grossman. 2024. CodeAid: Evaluating a Classroom Deployment of an LLM-based Programming Assistant that Balances Student and Educator Needs. In *Proceedings of the CHI Conference on Human Factors in Computing Systems (CHI '24)*. ACM, 1–20. doi:10.1145/3613904.3642773

[21] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2023. Is Your Code Generated by ChatGPT Really Correct? Rigorous Evaluation of Large Language Models for Code Generation. In *Thirty-seventh Conference on Neural Information Processing Systems.* https://openreview.net/forum?id=1qvx610Cu7

[22] Wenhan Lyu, Yimeng Wang, Tingting (Rachel) Chung, Yifan Sun, and Yixuan Zhang. 2024. Evaluating the Effectiveness of LLMs in Introductory Computer Science Education: A Semester-Long Field Study. In *Proceedings of the Eleventh ACM Conference on Learning @ Scale (L@S '24)*. ACM, 63–74. doi:10.1145/3657604.3662036

[23] Geoff Norman. 2010. Likert scales, levels of measurement and the "laws" of statistics. *Advances in health sciences education* 15, 5 (2010), 625–632.

[24] OpenAI. 2024. GPT-4o System Card. https://openai.com/index/gpt-4o-system-card/. Accessed 2025-09-29.

[25] OpenAI. 2024. Learning to Reason with LLMs. https://openai.com/index/learning-to-reason-with-llms/. Accessed 2025-09-29.

[26] OpenAI. 2025. Introducing OpenAI o3 and o4-mini. https://openai.com/index/introducing-o3-and-o4-mini/. Describes o4-mini as optimized for fast, cost-efficient reasoning.

[27] OpenAI. 2025. Reasoning Best Practices. https://platform.openai.com/docs/guides/reasoning-best-practices.

[28] Mrigank Pawagi and Viraj Kumar. 2024. Probeable Problems for Beginner-level Programming-with-AI Contests. In *Proceedings of the 2024 ACM Conference on International Computing Education Research - Volume 1* (Melbourne, VIC, Australia) *(ICER '24)*. Association for Computing Machinery, New York, NY, USA, 166–176. doi:10.1145/3632620.3671108

[29] Reinhard Pekrun, Anne Frenzel, Thomas Goetz, and Raymond Perry. 2007. The control-value theory of achievement emotions: An integrative approach to emotions in education. *Publ. in: Emotion in education / ed. by Paul A. Schutz and Reinhard Pekrun. Amsterdam : Academic Press, 2007, pp. 13-36* (01 2007).

[30] James Prather, Brett A. Becker, Michelle Craig, Paul Denny, Dastyni Loksa, and Lauren Margulieux. 2020. What Do We Think We Think We Are Doing? Metacognition and Self-Regulation in Programming. In *Proceedings of the 2020 ACM Conference on International Computing Education Research* (Virtual Event, New Zealand) *(ICER '20)*. Association for Computing Machinery, New York, NY, USA, 2–13. doi:10.1145/3372782.3406263

[31] James Prather, Paul Denny, Juho Leinonen, Brett A. Becker, Ibrahim Albluwi, Michelle Craig, Hieke Keuning, Natalie Kiesler, Tobias Kohn, Andrew Luxton-Reilly, Stephen MacNeil, Andrew Petersen, Raymond Pettit, Brent N. Reeves, and Jaromir Savelka. 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 Computer Science Education* (Turku, Finland) *(ITiCSE-WGR '23)*. Association for Computing Machinery, New York, NY, USA, 108–159. doi:10.1145/3623762.3633499

[32] Victor-Alexandru Pădurean, Paul Denny, Alkis Gotovos, and Adish Singla. 2025. Prompt Programming: A Platform for Dialogue-based Computational Problem Solving with Generative AI Models. In *Proceedings of the 30th ACM Conference on Innovation and Technology in Computer Science Education V. 1* (Nijmegen, Netherlands) *(ITiCSE 2025)*. Association for Computing Machinery, New York, NY, USA, 458–464. doi:10.1145/3724363.3729094

[33] 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. doi:10.1145/990555.990598

[34] Xiaochen Wang, Junqing He, Zhe yang, Yiru Wang, Xiangdi Meng, Kunhao Pan, and Zhifang Sui. 2024. FSM: A Finite State Machine Based Zero-Shot Prompting Paradigm for Multi-Hop Question Answering. arXiv:2407.02964 [cs.CL] https://arxiv.org/abs/2407.02964

[35] Steven L. Wise and Christine E. DeMars. 2005. Low Examinee Effort in Low-Stakes Assessment: Problems and Potential Solutions. *Educational Assessment* 10, 1 (2005), 1–17. doi:10.1207/s15326977ea1001_1