

Porpoise: An LLM-Based Sandbox for Novices to Practice Writing Purpose Statements

Shriram Krishnamurthi

Brown University
Providence, RI, USA
shriram@brown.edu

Thore Thießen

University of Münster
Münster, Germany
t.thiessen@uni-muenster.de

Jan Vahrenhold

University of Münster
Münster, Germany
jan.vahrenhold@uni-muenster.de

Abstract

Software developers have long emphasized the need for clear textual descriptions of programs, through documentation and comments. Similarly, curricula often expect students to write purpose statements that articulate in prose what program components are expected to do. Unfortunately, it is difficult to motivate students to do this and to evaluate student work at scale.

We leverage the use of a large language model for this purpose. Specifically, we describe a tool, PORPOISE, that presents students with problem descriptions, passes their textual descriptions to a large language model to generate code, evaluates the result against tests, and gives students feedback. Essentially, it gives students practice writing quality purpose statements, and simultaneously also getting familiar with zero-shot prompting in a controlled manner.

We present the tool’s design as well as the experience of deploying it at two universities. This includes asking students to reflect on trade-offs between programming and zero-shot prompting, and seeing what difference it makes to give students different formats of problem descriptions. We also examine affective and load aspects of using the tool. Our findings are somewhat positive but mixed.

CCS Concepts: • Social and professional topics → Computing education.

Keywords: purpose statements, docstrings, large language models, introductory programming, zero-shot prompting

ACM Reference Format:

Shriram Krishnamurthi, Thore Thießen, and Jan Vahrenhold. 2025. Porpoise: An LLM-Based Sandbox for Novices to Practice Writing Purpose Statements. In *Proceedings of the 2025 ACM SIGPLAN International Symposium on SPLASH-E (SPLASH-E ’25), October 12–18, 2025, Singapore, Singapore*. ACM, New York, NY, USA, 16 pages. <https://doi.org/10.1145/3758317.3759683>



This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License.

SPLASH-E ’25, Singapore, Singapore

© 2025 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-2142-7/25/10

<https://doi.org/10.1145/3758317.3759683>

1 Introduction

Developers have long emphasized the need for clear prose to accompany code: in API documentation, as docstrings, in commit logs, design trackers, engineering blogs, etc. [29, 31, 32]. In other words, programmers craft prose as much as code, and must do it well.

Mirroring this, many curricula also expect students to articulate key parts of their programs. For instance, *How to Design Programs* [12] (HTDP) asks students to write a *purpose statement (PS)* for every function. The *PS* gives a high-level behavioral description of the function. (In this paper we use *PS* instead of “docstring” both to refer to the *intent* rather than *construct*, and because that is the term the classes used.)

Unfortunately, grading a *PS* manually is slow and expensive. Worse, *PS*s are hard to *motivate*. Unlike types, tests, and code, they have no executable consequences. In our significant experience teaching this material, students tend to be lackadaisical about them, thus missing an opportunity to practice a long-term skill. Large language models (LLMs) offer the opportunity to change this.

It might be tempting to use an LLM as a textual engine, asking it to compare the student’s *PS* against an idealized one or to reformulate it [7]. However, this can be fragile, especially if the LLM does not understand technical or problem-specific vocabulary, and so on. It would also be dangerous if the LLM provided misleading feedback.

Instead, we use the LLM *only to generate code*. This has multiple benefits:

- Ultimately, code is what most students are motivated to produce, so this aligns well with their own desires.
- Code can be unambiguously evaluated using tests and other mechanisms.
- The ability to produce correct, working programs just from text introduces students to some of the power of LLMs.
- LLMs show how we can “mechanize” *PS*s, putting them on a much more elevated footing than before.
- This is a form of *zero-shot prompting*—using an LLM without problem-specific training—which is a useful new skill.

The rest of this paper works through the details of this idea. The broader research agenda is effectively:

- What is a good way to use LLMs to mechanize *PS*s?

- What different forms of problem description can we use?
- What affective and load effects does this engender?

We address very specific research questions in §7.

This work focuses at the *introductory* level, where friendly tool support especially matters. Those who are nervous about the subject often find unstructured exploration intimidating and may drop out. Thus, they most need scaffolding of the kind we provide.

2 Mechanics of PORPOISE

Concretely, we describe PORPOISE, our tool¹ that implements the ideas in this paper. Along the way, we highlight various design decisions embedded in the tool. These are denoted ◀ *in this manner* ▶.

2.1 Problem Description

After log-in, students chose a problem from a drop-down:

Choose problem:

The author of the problem has complete latitude in how they choose to name the problem. In the example above, the author has chosen to give the function the extremely generic name *a*, thereby completely obscuring its purpose. The author could instead have given a very helpful mnemonic name. ◀ *The choice of name matters* ▶; see §3.

Upon choosing a problem, the student is given a description of the problem for which they have to write a *PS*.

This should immediately give the reader pause: *how do we describe the problem?* If the problem description is given as prose, then it can already be used as a *PS*! Indeed, if doing so does not produce correct code, arguably the description is to blame.

We therefore need a different “vocabulary” for describing a problem. We are inspired by prior works [25, 33] that use examples/unit tests for problem *comprehension*: ◀ *we use them for problem description* ▶. Thus, PORPOISE shows students a set of examples, like this sample for an addition function.

Inputs		Output
2	3	5
-1	-2	-3
7	-2	5
-2	2	0

This representation has the benefit of being concrete and of dodging linguistic concerns (e.g., students with different native languages). Of course, many functions could meet a particular definition, so the student has to infer the right intent for the function; our actual examples have many more rows, and we discuss this issue in our findings. Finally, it connects *PSs* and programming to *testing*, which is especially important in some curricula, such as those following HTDP.

¹The name “PORPOISE” is inspired by the sea turtle in *Alice in Wonderland* [5], who is called a Tortoise because he “taught us”. Our tool hopes to teach students to write better “porpoise statements” [sic].

2.2 Purpose Statement Processing

Given these examples, the student writes a *PS* in a text box. There is no a priori limit on its length. This text is sent to an LLM, specifically (in this paper) OpenAI’s ChatGPT model gpt-3.5-turbo. ◀ *The choice of models matters* ▶; see §9.

PORPOISE opens the ChatGPT system prompt with the following text, which was arrived at after several iterations (the choice of programming language is explained in §4):

You are a programming assistant that generates programs in the Racket programming language. Your response should contain *only* a Racket program. It should NOT include anything else: explanation, test cases, or anything else. The output should be a Racket function that can be evaluated directly. It should begin with "(define" and end with ")", e.g., (define (f x) x), but replaced with the actual function you produce.

Crucially, PORPOISE asks ChatGPT to generate ◀ *not one but three implementations* ▶. This is because it is possible to get lucky with a weak *PS* once, but it is hard to get lucky multiple times. In particular, PORPOISE ◀ *uses a high “temperature” value of 0.8* ▶. The temperature is, very roughly, an indication of how much randomness one wants in the output; for ChatGPT, these values range between 0 (maximal determinism) and 1 (least determinism). The high temperature therefore encourages ChatGPT to interpret the prompt broadly. Thus, success requires robust prompts, not just getting lucky.²

Alternatively, one could also use multiple LLM engines. Since their underlying language models are not the same, that itself will induce alternate interpretations. In either case, the non-determinism models real-world communications between humans in software development teams and thus underlines the need to formulate *PSs* as precisely as possible to be understood by different humans (or LLMs for that matter).

2.3 Result Evaluation and Presentation

The three programs generated by the LLM are run against test suites. ◀ *The test suite is broken up into a public suite and a private one.* ▶ The public suite is exactly the examples that students were shown when they asked for a problem description. The private suite, however, stays hidden. The reason for the private tests is to avoid overfitting, just like the “training” versus “evaluation” sets used in machine learning.³ Thus, a student might see output like this:

²This number was chosen empirically: by trying different *PSs* and seeing what temperature value gave enough but not too much diversity of programs. This will naturally depend on the specific LLM used, and so on.

³In our studies, we saw very few instances of students pasting the provided examples into the prompt. Doing so would anyway not help, because an over-specific function would fail the private suite. The private suite *forces* the student to generalize.

Implementation	Private				Public					
					5	5	5	5	-3	0
1	success	success	success	success	success	success	success	success	success	success
2	success	success	success	success	success	success	success	success	success	success
3	success	success	success	success	success	success	success	success	success	success

where the headers in the Public columns are the expected outputs. This is of course the ideal case: all three implementations passed all tests. However, there are several kinds of unhappy outcomes:

wrong answer For a *public* test, if the program produces the wrong answer, PORPOISE shows the answer that it produced. This lets a student try to infer what the generated code actually does, so they can adjust their *PS*.

failure For a *private* test, if the program produces the wrong answer, there is no point showing the answer because the student has nothing to compare it to. Thus, they are just told that there was a test failure.

error The code produced by ChatGPT produced an error while running. Though this can happen for many reasons, often it's because there's a mismatch in type from what the tests were expecting and what ChatGPT produced. Typically, it means the student should add more to the *PS* about the number and type of expected arguments and the type of the result.

Thus, a student might instead see a result like this:

Implementation	Private				Public					
					5	5	5	5	-3	0
1	failure	error	failure	failure	success	3/2	success	-7/2	1/2	success
2	failure	success	success	success	2/3	success	0	success	success	-1
3	success	success	failure	failure	success	success	success	-7/2	1/2	success

The student can see that the first implementation produced the value 3/2 for input pair (3, 2), where it was expected to produce 5. Based on these and other values, this implementation seems to be dividing instead of adding. (We wrote a *PS* to suggest division in some implementations, so as to generate the above grid.)

There are a few more output conditions that are not particularly interesting here: ChatGPT took too long to respond, ChatGPT produced output that was simply not valid, and so on. These correspond to various systems and LLM conditions and were explained to students in a manual [20].

2.4 When to Stop?

Ideally, students would like to get an “all-success grid”. However, ChatGPT is not deterministic and we have a high temperature setting, so even re-running an all-success *PS* does not guarantee the same result because different code is generated the next time. And vice versa: if the student feels they have a good *PS*, simply re-running it can achieve all-success. This by itself, is a very important practical lesson to learn for students who interact with LLMs.

In our manual, students were warned that (a) multiple iterations were necessary, and (b) sometimes, even after several

tries, it can be difficult to achieve all-success. They were told to instead stop once they had made a good-faith effort.

2.5 Do Students See Source?

A natural question is, does PORPOISE show students the code generated by ChatGPT? ◀ *It does not, for three reasons:* ▶

1. It is very difficult to get ChatGPT to generate code in only the sub-language [14] being used in class. The result is that the generated code (a) often violates the course's style guidelines, but even more importantly (b) sometimes uses constructs students have not seen, so they might have trouble understanding the program at all.
2. By permitting it to use the full Racket language, rather than a sub-language, we maximize ChatGPT's chances of finding a solution.
3. Finally, this lets us put the emphasis on *purpose statements* rather than on code. The *PS* is the only mechanism the student has for steering the LLM to all-success.

Note that even courses that do not use formal sub-languages, as provided by Racket, often still have *implicit* sub-languages: e.g., the course has introduced only one kind of loop or not yet shown exceptions.

We could have chosen a different route. For instance, we could have parsed the resulting program to check whether it fit the stylistic rules and sub-language. But if it did not, the student would have seen a somewhat mysterious failure. We believe it is an open question how to show generated code *while adhering to stylistic constraints and sub-languages*.

2.6 Faulty Code as Partial Specification

For Round 2 (§7), we added a new feature to PORPOISE. If specified for a problem (§3), a student is shown not only the examples suite but also a *somewhat incorrect program* as part of the initial display. (In this case, the examples represent addition, but the buggy implementation performs subtraction instead.)

Buggy Implementation

Inputs		Output
2	3	5
-1	-2	-3

Our goal was to see ◀ *whether this would help or hurt students* ▶: it could help by giving them a general sense of the program, but it could also hurt by getting them to fixate on incorrect features (and neglecting the examples). We evaluate its effect in §7.4.

In our current implementation, this faulty program is not shown every time. Instead, the software shows it half the time and in alternation:⁴ e.g., a student given six problems will

⁴Formally, PORPOISE chooses based on a combination of the problem's position and the last character of the student's identifier. We choose the *last* character because this is much more likely to be uniformly distributed,

either see the faulty code (if provided) on the first, third, and fifth problems, or on the second, fourth, and sixth problems. This is mainly for research reasons: to see whether it makes any difference to how quickly students solve a problem. This is easy change, and the authors would be happy to help users who want a different policy.

3 Problem Authoring

Instructors can easily define their own problems. These are stored in a separate specification file, so that instructors do not need to touch PORPOISE source in any way. The specification is a convenient semi-structured format (currently s-expressions, but we can easily support other syntaxes like JSON and XML on demand). See appendix A for an example. (All supplementary material can be found at <https://cs.brown.edu/research/plt/dl/splash-e-2025/>.)

Each problem specification consists of a private name, synthesized name, public test suite (“training” set), private test suite (“evaluation” set), and optional bad implementation.

The private name (e.g., “addition”) reflects the true purpose of the problem. It is never shown to the student, or even sent over the network (where a knowledgeable student can use the browser’s inspection facilities to see the name).⁵ It only appears in logs, where it makes reading and processing log files much easier.

The synthesized name (e.g., “a”) is the name both shown in the menu and sent to the LLM. It is *important for the name to not be suggestive of the purpose*, because using suggestive names tends to cause the LLM to “guess” the desired function irrespective of the student’s PS. This is problematic in two ways. Most obviously, it compensates for poor student PSs. More subtly, it can cause the LLM to generate code that it has been trained on but does not exactly match the problem statement. For instance, we have seen the use of a suggestive name cause ChatGPT to synthesize an imperative rather than functional solution, causing the generated code to fail all the tests for no fault of the PS. (The problem of LLM guessing doesn’t end with the name; see §7.2.)

4 Choice of Language

As the system prompt in §2.2 shows, PORPOISE generates Racket [13] code. While there is no *fundamental* reason for this—in principle, PORPOISE can be made to work with any language with suitable runtime control—we chose to use Racket for the following reasons:

- In our experience, at this point LLMs have been trained *extremely* well on code in commonly-used languages like Python and Java. *It is therefore quite difficult to*

whereas the initial characters often have special patterns. Of course, if students are assigned random identifiers, then these can be designed to ensure such a distribution.

⁵All evaluation happens on a remote sever. The system is careful to not send any data that could be exploited by a user familiar with the browser’s facilities.

get them to not infer intent and generate correct code. ▶ This makes them worse than useless for our application, because they would give students false confidence in how effective their PSs are. Picking an overly obscure language means LLMs sometimes fail to generate even syntactically correct code. In our experiments, Racket has served as a useful mid-point.

- We need to protect the evaluation server from accidents and malice. For instance, a student could see a problem asking them to add two numbers, but could write “delete all the files”—for which ChatGPT will generate perfectly accurate code. *Protecting against such attacks is critical* ▶. PORPOISE exploits Racket’s sandbox library. That is, a student can write such a PS, ChatGPT will generate accurate code, but when PORPOISE runs it, it has no effect.
- In addition, *we must also protect against programs that run for a long time or even forever* ▶, whether intentionally or inadvertently. Racket’s engine abstraction addresses this problem very nicely.

The latter two can be accomplished by other means (e.g., some systems use Docker to achieve similar kinds of isolation), but puts some burden on installation for some users.

Conceptually, there is no reason PORPOISE could not use some other programming language. Indeed, the generated language should match what students are learning even if they do not see the generated code. That way the vocabulary is interpreted accurately. For instance, some students used terms like “drop elements” in their PS. In Racket, it usually means a *functional* update, whereas in Python or Java, it may mean an *imperative* one; using the wrong language would cause tests to fail for no fault of the PS.

5 Configuring and Running PORPOISE

PORPOISE is designed to be easy to install and run. The software is available from <https://github.com/shriram/porpoise>. The instructor just needs an OpenAI (or other) access token and a recent version of Racket, which installs seamlessly on Linux, MacOS, and Windows. The repository gives full instructions and offers an *optional* Dockerfile for those who want one. PORPOISE has two configurations options:

- An OpenAI response timeout can be tweaked to reflect server performance and number of students.
- A run of PORPOISE can be configured to take a specific user login through a URL parameter. Students can thus go to a form that performs institutional login, maps their real identity to an anonymous one, and initiates PORPOISE with the latter. This both saves PORPOISE from having to handle each institution’s authentication mechanism and, critically, anonymizes its logs. The initial form can even feature an initial survey. We exploit this combination in §7.1.

Indeed, we can report that users not connected to this paper have been able to successfully install and run PORPOISE for their students without any assistance from the authors.

6 Intervention 1: US CS 1.5

For this paper, we conducted two interventions which differed along several dimensions: country, student background, problem sets, etc. The first deployment was mainly formative, while the second drew on what we learned and was structured more as a formal research project. Thus, instead of artificially unifying them, we present them separately below and in §7, respectively.

6.1 Institution and Students

This intervention, from July–August 2023, reports on student performance at a highly-selective US university. Students were in the process of placing into an accelerated introductory course. Most of the students had some prior programming experience from high school. Therefore, most of them are comparable to students who have finished a CS1 course (though a few had no prior programming experience at all). We therefore label this group “US CS 1.5”.

The process was conducted in Racket using HTDP. By the time students arrived at this task, they had done four homeworks, covering atomic data (numbers, strings, Booleans, images); functions over atomic data; structures; functions that consume lists of atomic data and structures; functions that consume and produce such lists; and basic higher-order functions (which were not used in this assignment). Essentially, they had seen and practiced material that would teach them how to write these functions by hand for themselves.

Per the institution’s rules, this work does not fall under Human Subjects Research. Nevertheless, students had answered the question, “Can we use your work in educational reports/research?” Of 88 students who did this task, 83 confirmed the use of their data. We limit ourselves to data from students who gave consent.

PORPOISE was used as described in §2, with the exception of §2.6 (which had not yet been conceived).

6.2 Problems Used

Students were given a total of 11 problems, of which one, adding two numbers, was part of the manual. All 11 are as summarized in fig. 1. The actual problems had several examples (which can be seen in the repository); due to space limitations, here we show just one for illustration. These are routine CS1 programs in a functional programming course, akin to the “finger exercises” in *The Little Schemer* [16].

6.3 Student Performance

Usage. Students collectively evaluated 3218 PSs. All told, the cost came to about USD 13.50, paid from an institutional account (i.e., free to students).

Scores. To summarize student performance, each PS evaluation was given a numeric score. Each success was given one point, each error lost a point, and to penalize overfitting, each failure lost two points. The score across all tests was divided by the number of tests. Thus, a perfect score would be 1. In general we would hope to see scores of at least 0, but a negative score is possible.

For each student, we considered their *best* performance (which may not necessarily be their *last* evaluation). On all problems other than RL, students had a median best score of 1. Students also had a mean best score over 0.9 on all but RL (0.57), ALT2 (0.83), RS (0.87), and TS (0.58). We discuss what we saw in more detail:

- More than half the class was able to fully solve just about every problem (median score of 1), and the high averages indicate that students were able to infer the problem correctly from the examples and produce reasonable PSs.
- Familiarity and vocabulary help! Students were familiar with `append`, which is what `J` matches. Some recognized it and used the term. In contrast, most did not know that `z` corresponded to `zip`. Simply using that one word as the “purpose statement” would have sufficed, but most students had to provide a much longer description.
- RL was obviously problematic. The difficulty here is that, while ChatGPT is often able to handle instructions about the *first* element of a list, it could not reliably handle “last”. Many students resorted to reversing the list, removing the first matching instance, and then reversing it again.
- TS posed problems because of the difficulty of getting ChatGPT to “parse” the input list into a sequence of sub-lists, each of which needed to be added.
- NNA scored well but also created problems. ChatGPT often interprets “non-negative” as “positive”, thereby producing the wrong results on some inputs.
- Sloppy words can cause problems. Colloquially, it is perhaps reasonable to think of DBL as “doubling” each element of a list. However, ChatGPT invariably interprets that as multiplying each value by two. That means `(1 2 3)` would turn into `(2 4 6)`, which is not at all the desired answer.
- Most of all, one would expect ALT1 and ALT2 to have almost identical descriptions. However, while almost any description of “alternating elements” produced ALT1, it was surprisingly difficult to persuade ChatGPT to produce an implementation corresponding to ALT2. That is, one problem seems to be genuinely harder than the other!

Uses per Problem. Another measure of problem difficulty is how many times students evaluated PSs for it. We especially examine the *last* point at which they attained their

Short Name	Description	Illustrative Example
ADD	add two numbers	$2\ 3 \rightarrow 5$
NNA	average of the non-negative numbers in a list	$'(1\ 2\ -1\ -4\ 4\ 9) \rightarrow 4$
DBL	duplicates every list value	$'(1\ 2\ 3) \rightarrow '(1\ 1\ 2\ 2\ 3\ 3)$
RL	removes the last instance of the second parameter from a list	$'(x\ y\ x)\ 'x \rightarrow '(x\ y)$
ALT1	keeps alternating elements (from the first one)	$'(1\ 2\ 3\ 4\ 5) \rightarrow '(1\ 3\ 5)$
ALT2	keeps alternating elements (from the second one)	$'(1\ 2\ 3\ 4\ 5) \rightarrow '(2\ 4)$
RS	running sum	$'(1\ 2\ 3\ 4\ 5) \rightarrow '(1\ 3\ 6\ 10\ 15)$
A3	average-of-3	$'(1\ 3\ 5\ 1\ 6\ -4) \rightarrow '(3\ 3\ 4\ 1)$
TS	sum of elements between zeroes	$'(1\ 2\ 1\ 0\ -1\ -1\ 0) \rightarrow '(4\ -2)$
J	append two lists	$'(a\ b)\ '(c\ d) \rightarrow '(a\ b\ c\ d)$
Z	zip two lists	$'(a\ c)\ '(b\ d) \rightarrow '(a\ b\ c\ d)$

Figure 1. Intervention 1: Problem Descriptions.

highest score (students often undertook 10-20% more attempts before stopping). Again, we summarize.

The average number of tries to get their best score varies enormously. Familiar problems like ADD and J average around two tries. It is under 10 for most problems except RL (34.43), ALT2 (10.61, against 8.71 for ALT1s), and TS (20.73).

The contrast between ALT1 and ALT2 is especially interesting. Students made a mean of 5.94 (median 5) *PS* evaluations for ALT1 before their *first PS* evaluation for ALT2. That means, when they began to work on ALT2, they had a fairly good *PS* already, which needed only slight revision. We would then expect to see them need very few tries for ALT2. The fact that they needed more tries for ALT2 than for ALT1 shows that these two very similar problems were, in fact, very different to ChatGPT.

Purpose Statement Lengths. Finally, we ask: how long do student *PS*s tend to be? This is a difficult to summarize meaningfully because each student writes many *PS*s. Instead, we examine one simple and meaningful statistic: per problem, what was the length in characters of their *most successful* attempt (the same ones we have considered above)?

The median lengths are 42 (for ADD) to 135 (for TS) characters. A median-length example for ADD is “takes in two numbers and returns their sum” and for TS is “given a list of numbers, returns a list of numbers where each number is the sum of all numbers including and preceeding a 0 in the list”. An author manually reviewed the *PS*s and observed:

- Students put genuine effort into writing detailed *PS*s.
- For some problems, a purely declarative *PS* does not suffice; it needs to procedurally guide the LLM, as described above for RL.
- Similar problems that pose different difficulty to an LLM, like ALT1 and ALT2, also manifest as differences in *PS* length (median 84 versus 99).
- Even in cases where a function had a short name known to students (like append for J), they often either did not recognize or use it and instead wrote detailed textual descriptions (for J the median is 69).

The Problem with RL. The data above show RL to be a significant outlier. Students were also hurt by a mistake by the instructor. One of the tests had a flaw, so it would always produce an “error” output. The data used above correct for this bug by removing these “errors” from the grading calculation, but clearly the presence of this bug impacted students in other ways also (with many expressing frustration with this problem on the course forum).

6.4 Student Reflections

After solving these problems in PORPOISE, students were asked to implement the same functions manually (in Racket). They were then asked to reflect on the differences between their experiences using the two approaches. Students were told there were no “right answers”, and asked to present their thoughts and preferences.

The instructor read and coded all the student responses. Their comments largely made the following observations:

- They found the process of using PORPOISE interesting and novel, and largely satisfying.
- They found it difficult to guide ChatGPT when it would not recognize what a student thought was already a quality *PS*.
- They found manual coding more time-consuming. This was in part due to lack of familiarity with Racket’s libraries or not being allowed to use them (or even thinking some concepts—like list indexing—don’t exist in Racket, even though they do).
- They appreciated the predictability of manual programming. Many noted liking having control over the task, being able to estimate how much was left to do, and so on.
- Many students observed that if a *PS* yielded a partially-correct solution, simply adding to it did not necessarily improve the solution. Often they had to go for shorter rather than longer, more detailed descriptions.
- A few students picked up on the declarative versus procedural distinction, noting that to get working code, it

was often necessary to write something they felt was not a *purpose statement* (as opposed to an *implementation plan*: e.g., double-reversal).

- Most students who stated a preference felt they would use PORPOISE first, but if they didn't succeed—implicitly defined here as passing the two test suites—in a few tries, would switch to manual programming. (They implicitly assumed that a “production” version of PORPOISE would show them the generated code.)

Interestingly, many implicitly viewed PORPOISE as a general-purpose code-generator. However, PORPOISE does not work that way! It can only help with functions *for which it has a test suite*. This was a useful nudge that writing one's own examples and tests is useful when interacting with LLMs, much like writing *acceptance tests* to evaluate products built by a third-party in commercial settings. On the class forum, the instructor made a comment about this, which seemed to be received well.

7 Intervention 2: Europe CS 1

As noted (§6), the second use of PORPOISE was structured in a more traditional research format. Details of the research questions are given in §7.4.

7.1 Institution and Students

The second intervention, from November–December 2023, reports on student performance at a selective European public university. The course was a regular *Introduction to Programming* course that covered both functional and object-oriented programming. In the first half of the semester (October 2023–February 2024), the course followed the HTDP book and thus matched the setting in §6; the intervention was placed in that half of the semester. PORPOISE was used as described in §2, including the feature in §2.6. In total, 296 out of 335 students actively participating in the course took part in the study. Additional details are given in appendix B.1.

7.2 Problems Used

First Round Problems. The first round of PORPOISE assignments was given after six weeks of the course. At this point, students had been exposed to Racket for four weeks and worked on assignments involving atomic data (numbers, strings, Booleans) and functions over atomic data.

The problems for this round (see Figure 2) were divided into three groups: the same warm-up (ADD) used in the first intervention, four (SORTDEC, DIGITS, PRODMAX, DIVS) involving conditionals, and two on strings (REM, PIGLET).

The problems on conditionals were complemented by a somewhat incorrect solution (following §2.6). The strings problems was designed so that students could not solve them using the language features they had seen so far; this way, we sought to find out whether students realized that providing meaningful *PSs* would empower them to “solve” assignments

that required more advance knowledge. (See appendix B.2.) A direct consequence of this decision, however, was that we could not possibly provide “buggy” implementations as a display option.

The final problem, PIGLET, is modeled after the well-known “Pig Latin” (https://en.wikipedia.org/wiki/Pig_latin) language game. This was chosen because this game is an in-class problem in the second half of the semester. However, almost any *PS*—no matter how incorrect—that was suggestive of the problem and used the exact suffix of Pig Latin (“-ay”) caused ChatGPT to generate a correct Pig Latin solution! We therefore came up with this equivalent, but sufficiently different, problem to ensure that the *PS*, and not prior knowledge about this problem, was used to generate the solution.

Second Round Problems. The second round of PORPOISE assignments was given after eight weeks of the course. At this point, students had been exposed to Racket for six weeks and—just like in §6.1—had homework assignments covering atomic data (numbers, strings, Booleans; they had not worked on images but these were not part of any PORPOISE assignment); functions over atomic data; structures; functions that consume lists of atomic data and structures; functions that consume and produce such lists; and basic higher-order functions. As with the first problem set, the last two problems were problems that could not yet be solved by the students manually because they had not been exposed to the relevant language features. (See appendix B.2.)

Problem Vetting. Before deployment, the public test suites were shown to both one author's research group and to another's Teaching Assistants (TAs), and both were asked to guess the function. We used their feedback to mould the tests. Of course, there is still a significant expertise gap between the respondents and typical CS1 students, and both research group members and TAs are likely to have more shared context and experience. Still, iterating with these groups helped us get some sense that the intents are guessable from the examples. Moreover, the iteration process both (a) got rid of any typos and other small flaws in the test suite, and (b) ensured that, by virtue of knowing the intended solutions, we had not inadvertently left out vital examples.

7.3 Student Performance

Usage. Collectively, students evaluated 4147 *PSs* in round 1 and 2492 in round 2. The total cost came to under USD 30 paid from an institutional account (i.e., free to students). We found corruption in the logs of 12 students, so these were excluded from the analysis.

Scores. We used the same scoring system as in §6.3. The scores are shown in fig. 4 and fig. 5 for the two rounds. Here, we see very different performance than in §6.3. We do again see conscientious *PSs* (confirmed by manual examination)—in this case, most *PSs* were written in German. Students

Short Name	Description	Illustrative Example
ADD	add two numbers	2 3 $\rightarrow$ 5
SORTDEC	sort three numbers into decreasing order	1 3 2 $\rightarrow$ "3 2 1"
DIGITS	concatenate two numbers such that the first number precedes the second	2 10 $\rightarrow$ 210
PRODMAX	return the product of two numbers if both are positive, else their maximum	2 3 $\rightarrow$ 6
DIVS	determine whether the smaller of two numbers divides the larger	12 3 $\rightarrow$ #t
REM	remove all characters from a string that occur in another string	"hey world" "aeiou" $\rightarrow$ "hy wrld"
PIGLET	convert a string to lowercase, add "-" to the end of the word, then moves the first letter to the end of this word and add "-Dude" if this letter is a vowel and "-Mate" else. Finally, make the first letter an uppercase letter	"Erick" $\rightarrow$ "Rick-E-Dude"

Figure 2. Intervention 2, Round 1: Problem Descriptions. Since students had not been exposed to lists, SORTDEC encoded the result in a string.

Short Name	Description	Illustrative Example
SUM-NON-NEG	compute the sum of all non-negative numbers in a list	'(-1 1 -3 3 4) $\rightarrow$ 8
ZIP	interleave two lists, start with the first element in the first list	'(a b c) '(x y z) $\rightarrow$ '(a x b y c z)
COUNTS	count the occurrences of a given symbol in a given list	'('a 'b 'a 'b) 'b $\rightarrow$ 2
BALANCED	determine whether a list contains as many strictly positive as strictly negative numbers	'(-1 0 1) $\rightarrow$ #t
TABLE-SUM	given a list of positive numbers separated by zeros, compute the sum of the numbers in each segment	'(1 0 1 1 0 2 1 0) $\rightarrow$ '(1 2 3)
AVERAGE-3	report the mean of the numbers in each sliding windows of size three in a list of at least three elements	'(4 0 2 7 9) $\rightarrow$ '(2 3 6)

Figure 3. Intervention 2, Round 2: Problem Descriptions. Some problems as the same as in fig. 1.

Short Name	Best Performance			Purpose Statement Length			Avg. Tries	
	Mean	Median	Std. Dev.	Mean	Median	Std. Dev.	To Best	Overall
SORTDEC	0.362	0.667	0.764	157.40	119	139.68	8.718	10.399
DIGITS	0.571	1.000	0.774	132.27	112	97.43	3.960	4.558
PRODMAX	0.448	1.000	0.793	176.53	155	119.96	4.014	4.489
DIVS	0.394	0.246	0.394	232.17	191	148.00	6.378	8.151
REM	-0.192	-0.333	0.499	201.61	173	122.32	7.918	10.746
PIGLET	-0.142	0.000	0.706	333.77	315	147.72	6.326	7.961

Figure 4. Intervention 2, Round 1: Best performance, purpose statement length, and average tries.

rarely averaged ten or more tries, in contrast to numbers of 20 and over for some problems in §6.3.

Most of all, we see quite poor scores on several problems: the negative means and medians close to zero are indicative of real difficulty with the problems. Manual examination of the *PSs* shows genuine attempts but with a variety of issues. In PIGLET, students did not generalize to all vowels, using only those in the training set. In REM, students tried to remove substrings instead of characters, or wrote *PSs* that only removed the first mach. For AVERAGE-3, students seemed

to entirely misunderstand the problem. Finally, for TABLE-SUM, students generally understood the problem, but their *PSs* could not be interpreted accurately enough by ChatGPT.

In short, we see many failure modes. In some cases, despite our vetting, our suite of examples was not enough to generate the right problem understanding. In other cases, students generally understood the problem but failed to generalize enough. Finally, in some cases the problem was with getting the LLM to translate their correct understanding into code.

Short Name	Best Performance			Purpose Statement Length			Avg. Tries	
	Mean	Median	Std. Dev.	Mean	Median	Std. Dev.	To Best	Overall
SUM-NON-NEG	0.807	1.000	0.501	155.23	120	117.85	2.165	2.287
ZIP	0.509	1.000	0.690	228.92	198	158.61	4.673	5.225
COUNTS	0.744	1.000	0.514	186.14	154	131.75	2.398	2.591
BALANCED	0.088	-0.111	0.614	206.71	177	143.27	3.167	4.264
TABLE-SUM	-0.110	0.000	0.739	232.02	194	156.29	5.894	8.230
AVERAGE-3	-0.484	-0.648	0.942	222.74	188	186.63	3.607	4.779

Figure 5. Intervention 2, Round 2: Best performance, purpose statement length, and average tries.

7.4 Research Questions and Their Answers

We now present research questions and brief answers to them. All the details of our analyses are given in the appendix (appendix B). Our questions arose from wondering: (a) whether prompting gives them additional “programming” capabilities; (b) whether PORPOISE changes their perception of PSs; (c) whether the flawed code examples (§2.6) made any difference; and (d) how PORPOISE impacted task complexity and affective aspects. The first two were motivated by the design of PORPOISE; the third to explore a design alternative; and the last are standard for many tools.

RQ-Can’t-Code: *Can students use PSs to successfully generate code for problems they don’t yet know how to program?*

All four problems that students would not have been able to solve with their programming knowledge are the problems on which they did poorest with PORPOISE. Therefore, this process did *not* showcase how an LLM could produce solutions that they could specify but not code. Instead, the results underline that programming has at least two important facets: understanding the programming language used, and mapping the task (e.g., “sliding window” from fig. 3) and problem-solving concepts to the features of said programming language.

While an LLM can remove most of the burden associated with the former aspect, successfully addressing the latter requires knowledge and expressiveness on the side of the prompter. PORPOISE thus does *not* increase performance by offloading cognitive tasks, which is a major risk to learning associated with many uses of generative AI [34]. We return to this point when we discuss consequences for program composition (“planning”) in §9.

RQ-Perception: *Does using PORPOISE change the students’ perceptions of purpose statements?*

Students were asked to rate their perceived usefulness of PSs before and after using PORPOISE. The ratings were very high (8/10) even before Round 1, so there was little room for actual improvement. We therefore saw no significant change after Round 1. Nevertheless, we did see a significant improvement after Round 2.

RQ-Prompt: *To what extent does it matter which kind of initial prompt they were given: just a table of examples, or flawed code (violating the examples) in addition to the examples?*

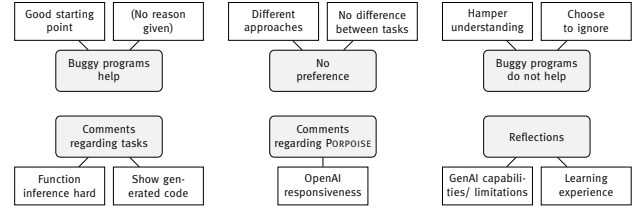


Figure 6. Intervention 2: Themes emerging from comments regarding preferences, tasks, PORPOISE, and reflections.

There was *no* significant difference favoring either showing or not showing a buggy program. Indeed, many students reported not even looking at the buggy programs.

RQ-Complexity: *What is the task-load complexity of PORPOISE?*

We used (most of) the NASA Task Load Index (NASA-TLX) [17]. Students (unsurprisingly) found PORPOISE not physically demanding, and not too temporally demanding. They rated their ability to successfully use the tool slightly more on the negative side, and effort slightly more on the “working hard” side.

RQ-Emotion: *What impact does PORPOISE have on emotions?*

Students were asked to compare using PORPOISE against traditional programming assignments, and assessed using the instrument by Lishinski, et al. [22]. Generally, PORPOISE evoked similar emotions to traditional programming assignments. In Round 2, we did see that PORPOISE induced less feeling of inadequacy/stupidity.

7.5 Qualitative Results

The post-intervention survey also contained two free-form text fields. The first asked students to comment on whether or not the “buggy programs” were considered helpful; the second asked for any comments they wanted to offer. The thematic analysis [3] of the 143 free-text responses found six themes to emerge from the data, shown in fig. 6. Students were split regarding the helpfulness of the “buggy programs”, with 34 responses commenting on the helpfulness, 40 responses commenting on the lack thereof, and 20 responses stating no preference. A recurring comment was that each setting resulted in a different problem-solving approach.

Other feedback included complaints about difficulty discerning the latent functions and on the capabilities and limitations of Generative AI, with students both enthusiastic and disillusioned. None felt negative about the experience, and many enjoyed “explaining the task” to the LLM.

8 Related Work

A large number of papers has been written in the past two years about LLMs in introductory programming. We refer to a recent survey [24] on these works.

With respect to the specific focus on presenting students with problem descriptions, asking them to use an LLM to produce solutions, and then assessing them against a test suite, a small number of other papers are related closely.

Denny et al. worked with problem descriptions presented as visual representations [9] or code snippets [10]. The visualizations in the first study presented examples from which students should infer the purpose of the code. In the second study, students were asked to give an “explain in plain English” [23] description of the code’s purpose, which—as in the first study—was then fed into an LLM to produce code which then was checked against tests. Thus, there are many similarities, but there are also important differences. In contrast to our paper, these do not:

- Study their tools across multiple institutions or countries of different flavors.
- Consider the impact of a misleading code prompt.
- Use temperature to tease out poorly-written prompts.
- Generate multiple implementations to give a better sense of prompt quality.
- Perform a qualitative analysis.
- Suppress generated code. Their student comments show some of the benefits and downsides of this.
- Avoid overfitting (as we do) through separate training and evaluation sets. Instead, they use *images* to present tests (hoping students will not type them in).
- Discuss visual accessibility; nor is it at all clear how they could support it. If the tests were presented as “alt text”, students could just as well copy that. But by not providing it, it renders screen-readers useless. Our work does not obscure the text in any way.

In addition, in §7 we discuss multiple study conditions, examine task load, and evaluate affective issues.

A very recent paper [18] by a related set of authors also reports on students’ reactions when interacting with LLM-generated feedback. Their work differs from ours in that the authors asked students to provide an “explain in plain english” description of an algorithm shown to them as correct C code, whereas we work with test cases and—at times—buggy code. They too note the mixed reaction of students, and identify concerns with the utility of their approach.

Finally, we have open sourced our work (§5) and others have been able to use it. We are not aware of the other

projects being available. Our system is also designed to work with different data privacy concerns, e.g., GDPR.

Our work is also related to that of Babe et al. [1, 2]. Their focus is on generating a *benchmark* of students writing prompts for LLMs. Their work uses the same format as PORPOISE to describe problems. The main difference is that their goal is to student-source prompts. (In principle, our data can be used to the same end.) Their paper does provide a sophisticated linguistic analysis of the student prompts. However, it does not provide the many educational elements or analysis described here.

It is possible we have missed other work in this space, which is after all fast-growing. That said, we believe our combination of: (a) a useful tool (that has been used by third-parties), (b) discussion of design considerations, (c) deployment across multiple institutions and countries, (d) analysis of both, with a detailed research evaluation of the second, and (e) treatment of a language other than Python, all make this work interesting.

9 Discussion

Choice of Models. Currently, PORPOISE uses OpenAI’s gpt-3.5-turbo model, which is now well behind the state-of-the-art. Practically speaking, weaker models are cheaper, which matters economically. But more importantly, it is critical that the model *not do too much*. Weak models are good at simulating a faulty listener, which forces the student to write as precise a *PS* as possible.

There is a non-trivial concern that as models get better and better, poorer and poorer *PSs* will still pass. In addition, the degree to which these LLMs have been trained on programming problems will also have an impact. Finally, there is little commercial incentive to providing lower-quality models, so they may disappear. Thus, over time a system like PORPOISE will need to self-host a model. The growing availability of so-called “open source” models that we can download and preserve therefore becomes paramount.

Prose Quality. LLMs comfortably handle prose that falls well short of high literary standards. Both in our (intentional) tests and in student work, we see no problem caused by typographical, grammatical, and punctuation errors. This robustness stands in interesting opposition to the extreme precision demanded by most programming languages, and should be impressed more on students.

Linguistic Diversity. ChatGPT, being trained on Web material amongst other content, appears to have a strong English bias [8] and recent studies have focused on linguistic diversity of both prompting and programming languages [26, 30]. However, for the level of our tasks, it has no difficulty with many widely-used languages. We have manually tested it on languages like French and Hindi *PSs* and it works fine. In addition, most of the *PSs* in §7 were in

German, and caused no difficulty. In contrast, providing a *PS* in Samoan (using Google Translate; the authors do not know Samoan) produced nonsensical output.

Instruction Sets. Traditional programming languages provide a clear instruction set to which the user maps their intent. In contrast, as problems like ALT2, RL, RS, and TS (§6.3) show, the LLM appears to have some latent “instruction set”. Determining this instruction set is made significantly complicated by many factors:

1. It must be discovered by experiment.
2. LLMs allow free-form natural language statement of intent, creating an indeterminate input space.
3. LLMs never reject an input.
4. This set will vary (and presumably almost always grow) over time.
5. This set will vary across LLMs, as a function of their training and tuning.

This stands in stark contrast to the fixed instruction sets of traditional programming. This type of discourse—which is common to all instructional approaches that directly interact with an LLM—can hinder a clear path from abstract to concrete and back [4] and, arguably, introduce new difficulties into programming.

Dependency on “Paradigms”. We have only used PORPOISE with students learning formal programming in Racket in a functional style. Functional programming lends itself especially well to unit-tests, where function arguments and return values fully capture the function’s behavior without having to write down messy input-output side-effects; in addition, there is no state from one run to the next.

We believe it would be significantly more difficult to create a system like PORPOISE in the latter setting. Indeed, it is not surprising that whereas curricula like HTDP emphasize writing examples and tests, many programs centered on imperative programming do not, or at least do not use *automated* testing: “testing” often means running a program and manually examining the output.

Consequences for Program Construction. One of our hopes is that PORPOISE will change the way students think about program construction. We often see students dive right into programming without thinking about first decomposing their task into smaller, semantically meaningful, manageable pieces. In contrast, in the 1980s, the computing education literature spent some effort thinking about program planning (e.g., [28]). This work did not see much more attention until roughly the past decade (e.g., [6, 15, 21]).

Traditional planning can still feel “unproductive” in a way that a *PS* does, because it doesn’t yield executable artifacts. If students realized, however, that a plan can then be automatically turned into code, that may cause them to change their approach to programming. Recent work on providing feedback on plans by leveraging LLMs takes this direction [27].

While planning usually refers to a phase before program authoring, a similar phenomenon occurs even while writing an individual function. In the midst of one, we may find we need to rely on some non-trivial piece of logic. This may either be something that would make the function too complicated to inline, or may even be something we don’t immediately see how to write.

HTDP encourages students to put such functions on a “wish list”. It recommends that every wish list entry have a name, function header, dummy body, and purpose statement, with the body to be completed later. The book says,

Writing down complete function headers ensures that you can test those portions of the programs that you have finished, which is useful even though many tests will fail. Of course, when the wish list is empty, all tests should pass and all functions should be covered by tests.

Now, however, we can have a define-by-wishing form that uses an LLM to fill in the definition. It may not be perfect, but it would at least make many more tests pass right away, so that the student knows they are on the right track! The student can also write definitional examples as part of the wish list—ideally both training and evaluation examples, mimicking what PORPOISE did for them.

10 Conclusion

Software engineers communicate in prose as much as code, but novice students, in particular, often overvalue code and undervalue prose. We created PORPOISE to put prose on a more operational footing in a setting students are already familiar with (*PSs*), while also giving them a controlled experience with LLMs.

Our experience is mixed: students were mostly successful constructing *PSs* that achieved correct results. The tool imposed low load, and reduced some feeling of stupidity or inadequacy, which plagues novices (and causes some of them to drop out of computing [11, 19, 22]). But it did not appreciably improve their appreciation for writing. Therefore, we believe much more work is needed in this space. We hope PORPOISE can serve as a starting point, especially for novices, and that the lessons of this paper spur further investigation.

Acknowledgments

We are grateful to Kathi Fisler for feedback, Arjun Guha for discussions, and Tara for being our first (trial) user. We thank our research group students and course teaching assistants for trying out the problems and giving feedback. We appreciate the thoughtful comments and suggestions for improvements by the reviewers. TABLE-SUM is inspired by Mike Clancy. SK acknowledges the US National Science Foundation grant DGE-2208731 for partial support.

References

- [1] Hannah McLean Babe, Sydney Nguyen, Yangtian Zi, Arjun Guha, Molly Q Feldman, and Carolyn Jane Anderson. 2023. StudentEval: A Benchmark of Student-Written Prompts for Large Language Models of Code. *arXiv:2306.04556* [cs.LG]
- [2] Hannah McLean Babe, Sydney Nguyen, Yangtian Zi, Arjun Guha, Molly Q. Feldman, and Carolyn Jane Anderson. 2024. StudentEval: A Benchmark of Student-Written Prompts for Large Language Models of Code. In *Findings of the Association for Computational Linguistics, ACL 2024, Bangkok, Thailand and virtual meeting, August 11–16, 2024*, Lun-Wei Ku, Andre Martins, and Vivek Srikumar (Eds.). Association for Computational Linguistics, Kerrville, TX, 8452–8474. [doi:10.18653/v1/2024.FINDINGS-ACL.501](https://doi.org/10.18653/v1/2024.FINDINGS-ACL.501)
- [3] Virginia Braun and Victoria Clarke. 2006. Using Thematic Analysis in Psychology. *Qualitative Research in Psychology* 3, 2 (2006), 77–101. [doi:10.1191/1478088706qp0630a](https://doi.org/10.1191/1478088706qp0630a)
- [4] Neil C. C. Brown, Feliene F. J. Hermans, and Lauren E. Margulieux. 2023. 10 Things Software Developers Should Learn about Learning. *Commun. ACM* 67, 1 (Dec. 2023), 78–87. [doi:10.1145/3584859](https://doi.org/10.1145/3584859)
- [5] Lewis Carroll. 1898. *Alice's Adventures in Wonderland*. Macmillan Company, New York, NY, USA.
- [6] Kathryn Cunningham, Barbara J. Ericson, Rahul Agrawal Bejarano, and Mark Guzdial. 2021. Avoiding the Turing Tarpit: Learning Conversational Programming by Starting from Code's Purpose. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems (Yokohama, Japan) (CHI '21)*. ACM Press, New York, NY, Article 61, 15 pages. [doi:10.1145/3411764.3445571](https://doi.org/10.1145/3411764.3445571)
- [7] Nicola Dainese, Alexander Ilin, and Pekka Marttinen. 2024. Can docstring reformulation with an LLM improve code generation?. In *Proceedings of the 18th Conference of the European Chapter of the Association for Computational Linguistics: Student Research Workshop*, Neele Falk, Sara Papi, and Mike Zhang (Eds.). Association for Computational Linguistics, St. Julian's, Malta, 296–312. <https://aclanthology.org/2024.eacl-srw.24/>
- [8] Paresh Dave. 2023. ChatGPT Is Cutting Non-English Languages Out of the AI Revolution. <https://www.wired.com/story/chatgpt-non-english-languages-ai-revolution/>.
- [9] 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, SIGCSE 2024, Volume 1, Portland, OR, USA, March 20–23, 2024*, Ben Stephenson, Jeffrey A. Stone, Lina Battestilli, Samuel A. Rebelsky, and Libby Shoop (Eds.). ACM Press, New York, NY, 296–302. [doi:10.1145/3626252.3630909](https://doi.org/10.1145/3626252.3630909)
- [10] Paul Denny, David H. Smith IV, Max Fowler, James Prather, Brett A. Becker, and Juho Leinonen. 2024. Explaining Code with a Purpose: An Integrated Approach for Developing Code Comprehension and Prompting Skills. In *ITiCSE '24: Proceedings of the 29th ACM Conference on Innovation and Technology in Computer Science Education Vol. 1*, Judithe Sheard and James Paterson (Eds.). ACM Press, New York, NY, 283–289. [doi:10.1145/3649217.3653587](https://doi.org/10.1145/3649217.3653587)
- [11] Maja Dornbusch and Jan Vahrenhold. 2024. “In the Beginning, I Couldn't Necessarily Do Anything With It”: Links Between Compiler Error Messages and Sense of Belonging. In *ICER '24: Proceedings of the 2024 ACM Conference on International Computing Education Research – Volume 1*, Paul Denny, Leo Porter, Maragret Hamilton, and Briana Morrison (Eds.). ACM Press, New York, NY, 14–26. [doi:10.1145/3632620.3671105](https://doi.org/10.1145/3632620.3671105)
- [12] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. 2018. *How to Design Programs* (second ed.). MIT Press, Cambridge, MA, USA. <http://www.htdp.org/>
- [13] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, Shriram Krishnamurthi, Eli Barzilay, Jay McCarthy, and Sam Tobin-Hochstadt. 2018. A Programmable Programming Language. *Commun. ACM* 61, 3 (March 2018), 62–71. [doi:10.1145/3127323](https://doi.org/10.1145/3127323)
- [14] Robert Bruce Findler, John Clements, Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, Paul Steckler, and Matthias Felleisen. 2002. DrScheme: A Programming Environment for Scheme. *Journal of Functional Programming* 12, 2 (2002), 159–182. [doi:10.1017/S0956796801004208](https://doi.org/10.1017/S0956796801004208)
- [15] Kathi Fisler. 2014. The Recurring Rainfall Problem. In *Proceedings of the Tenth Annual Conference on International Computing Education Research (Glasgow, Scotland, United Kingdom) (ICER '14)*. ACM Press, New York, NY, 35–42. [doi:10.1145/2632320.2632346](https://doi.org/10.1145/2632320.2632346)
- [16] Daniel P. Friedman and Matthias Felleisen. 1996. *The Little Schemer* (4 ed.). MIT Press, Boston, MA.
- [17] Sandra G. Hart. 2006. NASA-Task Load Index (NASA-TLX); 20 Years Later. In *Proceedings of the Human Factors and Ergonomics Society Annual Meeting*. SAGE Publications, Los Angeles, CA, 904–908. [doi:10.1177/154193120605000909](https://doi.org/10.1177/154193120605000909)
- [18] Chris Kerslake, Paul Denny, David H. Smith, Juho Leinonen, Stephen MacNeil, Andrew Luxton-Reilly, and Brett A. Becker. 2025. Exploring Student Reactions to LLM-Generated Feedback on Explain in Plain English Problems. In *Proceedings of the 56th ACM Technical Symposium on Computer Science Education V. 1 (Pittsburgh, PA, USA) (SIGCSE 2025)*. ACM Press, New York, NY, 575–581. [doi:10.1145/3641554.3701934](https://doi.org/10.1145/3641554.3701934)
- [19] Päivi Kinnunen and Beth Simon. 2010. Experiencing programming assignments in CS1: the emotional toll. In *Proceedings of the Sixth International Workshop on Computing Education Research, ICER 2010*, Michael E. Caspersen, Michael J. Clancy, and Kate Sanders (Eds.). ACM Press, New York, NY, 77–86. [doi:10.1145/1839594.1839609](https://doi.org/10.1145/1839594.1839609)
- [20] Shriram Krishnamurthi. 2023. Introduction to Porpoise. https://docs.google.com/document/d/1dHCev4LBFOQWKYQ1xA6i11uU7GCRHhAoMrVT2Ev_6wQ/edit?usp=sharing.
- [21] Shriram Krishnamurthi and Kathi Fisler. 2021. Developing Behavioral Concepts of Higher-Order Functions. In *ICER 2021: Proceedings of the 17th ACM Conference on International Computing Education Research*, Amy J. Ko, Jan Vahrenhold, Renée McCauley, and Matthias Hauswirth (Eds.). ACM Press, New York, NY, 306–318. [doi:10.1145/3446871.3469739](https://doi.org/10.1145/3446871.3469739)
- [22] Alex Lishinski, Aman Yadav, and Richard Enbody. 2017. Students' Emotional Reactions to Programming Projects in Introduction to Programming: Measurement Approach and Influence on Learning Outcomes. In *Proceedings of the 2017 ACM Conference on International Computing Education Research, ICER 2017*, Josh Tenenber, Donald Chinn, Judy Sheard, and Lauri Malmi (Eds.). ACM Press, New York, NY, 30–38. [doi:10.1145/3105726.3106187](https://doi.org/10.1145/3105726.3106187)
- [23] Raymond Lister, Beth Simon, Errol Thompson, Jacqueline L. Whalley, and Christine Prasad. 2006. Not Seeing the Forest for the Trees: Novice Programmers and the SOLO Taxonomy. In *Proceedings of the 11th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education, ITiCSE 2006*, Renzo Davoli, Michael Goldweber, and Paola Salomoni (Eds.). ACM Press, New York, NY, 118–122. [doi:10.1145/1140124.1140157](https://doi.org/10.1145/1140124.1140157)
- [24] James Prather, Juho Leinonen, Natalie Kiesler, Jamie Gorson Benario, Sam Lau, Stephen MacNeil, Narges Norouzi, Simone Opel, Vee Pettit, Leo Porter, Brent N. Reeves, Jaromir Savelka, David H. Smith, Sven Strickroth, and Daniel Zingaro. 2025. Beyond the Hype: A Comprehensive Review of Current Trends in Generative AI Research, Teaching Practices, and Tools. In *2024 Working Group Reports on Innovation and Technology in Computer Science Education (Milan, Italy) (ITiCSE 2024)*. ACM Press, New York, NY, 300–338. [doi:10.1145/3689187.3709614](https://doi.org/10.1145/3689187.3709614)
- [25] James Prather, Raymond Pettit, Brett A. Becker, Paul Denny, Dastyni Loksa, Alani L. 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, SIGCSE 2019, Minneapolis, MN, USA, February 27 - March 02, 2019*, Elizabeth K. Hawthorne, Manuel A. Pérez-Quinones, Sarah Heckman, and Jian Zhang (Eds.). ACM Press, New York, NY, 531–537. doi:10.1145/3287324.3287374
- [26] James Prather, Brent N Reeves, Paul Denny, Juho Leinonen, Stephen MacNeil, Andrew Luxton-Reilly, João Orvalho, Amin Alipour, Ali Alfageeh, Thezyrie Amarouche, Bailey Kimmel, Jared Wright, Musa Blake, and Gweneth Barbre. 2025. Breaking the Programming Language Barrier: Multilingual Prompting to Empower Non-Native English Learners. In *Proceedings of the 27th Australasian Computing Education Conference*. ACM Press, New York, NY, 74–84. doi:10.1145/3716640.3716649
- [27] Elijah Rivera, Alexander Steinmaurer, Kathi Fisler, and Shriram Krishnamurthi. 2024. Iterative Student Program Planning using Transformer-Driven Feedback. In *Proceedings of the 2024 on Innovation and Technology in Computer Science Education V. 1*. ACM Press, New York, NY, 45–51. doi:10.1145/3649217.3653607
- [28] Elliot Soloway. 1986. Learning to Program = Learning to Construct Mechanisms and Explanations. *Commun. ACM* 29, 9 (Sept. 1986), 850–858. doi:10.1145/6592.6594
- [29] Christoph Johann Stettina and Werner Heijstek. 2011. Necessary and neglected? an empirical study of internal documentation in agile software development teams. In *Proceedings of the 29th ACM International Conference on Design of Communication (Pisa, Italy) (SIGDOC '11)*. ACM Press, New York, NY, 159–166. doi:10.1145/2038476.2038509
- [30] Alaaeddin Swidan and Felienne Hermans. 2023. A Framework for the Localization of Programming Languages. In *Proceedings of the 2023 ACM SIGPLAN International Symposium on SPLASH-E*. ACM Press, New York, NY, 13–25. doi:10.1145/3622780.3623645
- [31] Christoph Treude, Justin Middleton, and Thushari Atapattu. 2020. Beyond accuracy: assessing software documentation quality. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (Virtual Event, USA) (ESEC/FSE 2020)*. ACM Press, New York, NY, 1509–1512. doi:10.1145/3368089.3417045
- [32] Akhila Sri Manasa Venigalla and Sridhar Chimalakonda. 2024. An exploratory study of software artifacts on GitHub from the lens of documentation. *Information and Software Technology* 169 (2024), 107425. doi:10.1016/j.infsof.2024.107425
- [33] 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 2019*, Robert McCartney, Andrew Petersen, Anthony V. Robins, and Adon Moskal (Eds.). ACM Press, New York, NY, 131–139. doi:10.1145/3291279.3339416
- [34] Lixiang Yan, Samuel Greiff, Jason M. Lodge, and Dragan Gašević. 2025. Distinguishing performance gains from learning when using generative AI. *Nature Reviews Psychology* 4 (June 2025), 2 pages. doi:10.1038/s44159-025-00467-5

Received 2025-06-30; accepted 2025-08-01