Iterative Student Program Planning using Transformer-Driven Feedback

Elijah Rivera
eerivera@brown.edu
Brown University
Providence, Rhode Island, USA

Alexander Steinmaurer
alexander.steinmaurer@tugraz.at
Graz University of Technology
Graz, Austria

Kathi Fisler
kfisler@cs.brown.edu
Brown University
Providence, Rhode Island, USA

Shriram Krishnamurthi
shriram@brown.edu
Brown University
Providence, Rhode Island, USA

ABSTRACT

Problem planning is a fundamental programming skill, and aids students in decomposing tasks into manageable subtasks. While feedback on plans is beneficial for beginners, providing this in a scalable and timely way is an enormous challenge in large courses.

Recent advances in LLMs raise the prospect of helping here. We utilize LLMs to generate code based on students' plans, and evaluate the code against expert-defined test suites. Students receive feedback on their plans and can refine them.

In this report, we share our experience with the design and implementation of this workflow. This tool was used by 544 students in a CS1 course at an Austrian university. We developed a codebook to evaluate their plans and manually applied it to a sample. We show that LLMs can play a valuable role here. However, we also highlight numerous cautionary aspects of using LLMs in this context, many of which will not be addressed merely by having more powerful models (and indeed may be exacerbated by it).

CCS CONCEPTS

• **Applied computing** → **Education**.

KEYWORDS

program planning; automated feedback; LLMs

ACM Reference Format:
Elijah Rivera, Alexander Steinmaurer, Kathi Fisler, and Shriram Krishnamurthi. 2024. Iterative Student Program Planning using Transformer-Driven Feedback. In *Proceedings of the 2024 Innovation and Technology in Computer Science Education V. 1 (ITiCSE 2024), July 8–10, 2024, Milan, Italy*. ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3649217.3653607

1 INTRODUCTION

In the context of programming, planning is the process of decomposing a program into subtasks and mapping out how the subtasks will fit together (aka compose) in a final solution. It creates a space for high-level thinking about a program prior to committing to details of code. A good plan should also help programmers focus on manageable-size chunks of a program, rather than trying to consider the entire program at once. In educational contexts, explicit planning might help students who feel overwhelmed by assignments to organize their thoughts and work, akin to using a to-do list to manage a complex event in daily life (as seen in fig. 2).

We focus on planning for students in early post-secondary computer science. Students at this level often struggle with programming; planning could potentially help them make progress in smaller steps. Students can also be reluctant to throw out a program and start over: knowing they are on a path to success before they commit to code could be useful. Ideally, we would test whether a proposed plan could solve the problem at hand prior to having a student code the solution. However, this is only useful if students can get close-to-immediate feedback on their plans, which is very difficult to do with manual grading alone, that too in large classes. Furthermore, traditional methods like software testing will not work on partial programs or textual plans.

The code-generation abilities of LLMs provide an exciting opportunity to support planning education at scale. If LLMs can translate a plan to code in a reasonable way, we could run that code against test suites and provide validation feedback, enabling the student to iteratively refine the plan until it is sufficiently viable. This experience report describes what happened when we tried this in a large novice introductory programming course.

2 RELATED WORK

Program Planning. Program planning finds its roots in plan composition work done by Spohrer and Soloway [26], who first posited that CS novices have an internal plan from which they are starting when solving a given programming problem. Attempts to investigate these plans first relied on interpreting them from student-created programs [9, 13, 26], and also from observing students during the programming process [4, 12, 22, 23]. Only a few prior projects have explicitly studied pedagogic tools for using plans, most notably Muller et al. on "pattern-oriented instruction" [19] and de Raadt et al.'s "strategy guide" [7]. The "patterns" and "strategies" identified by these two lines of work are very low-level imperative programming constructs.

In contrast, our recent prior work [17, 25] lays out a framework for a planning process that is a separate step with a deliverable before the programming process, and it does so with more abstract

ITiCSE 2024, July 8–10, 2024, Milan, Italy

ACM ISBN 979-8-4007-0600-4/24/07.
https://doi.org/10.1145/3649217.3653607

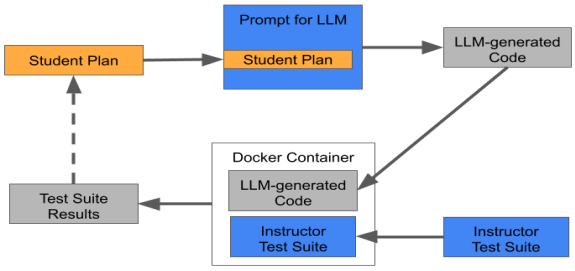

Figure 1: Tool and user workflow

programming idioms like higher-order functions. The vision for this project draws on these previous papers, and fits in line with the recent push for more explicit metacognition education in programming courses [1, 8, 18].

LLMs. Recent work has (somewhat) successfully prompted an LLM to produce both plans [14, 29], and code [5, 28], especially for programming problems typically found in CS classrooms [10, 11, 20, 21]. All that work has the goal of getting the LLM to produce increasingly *correct* plans/code; Badyal et al. [2] showed that at this point it is quite difficult to "convince" state-of-the-art LLMs to hallucinate incorrect answers. However, to give accurate and relevant feedback on student plans, we need the LLM to produce *syntactically correct but semantically wrong* code when a student submits an *incorrect* plan. We are not aware of any prior work that has this orientation.

3 TOOL DESIGN

Figure 1 shows the workflow that occurs under the hood. After selecting a problem to work on, the student writes a plan (in a textbox) and submits it for evaluation. Our tool constructs an LLM prompt that includes the student's plan, then runs the LLM-generated code against a staff-developed test suite. The tool then presents the testing results to the student, who can modify their plan and repeat the process. Our tool does not check any syntactic properties of the student's plan. We examine the structure of their plans in section 5 and discuss our reasons for not checking notation in section 6.

LLM and prompt configuration. The effectiveness of our workflow is highly dependent on both the LLM and prompt used. Two general failure modes are possible when using LLMs to generate code from plans. False negatives arise when the plan is adequate but the LLM-generated code lacks behaviors that the plan had covered. False positives arise when the plan is deficient but the LLM fills in enough details that the generated code passes the test-suite validation.

LLM. Initially, we attempted to build the tool atop OpenAI's GPT3.5 (`gpt-3.5-turbo-0613`). Our pilot experiments (not involving students) indicated that GPT3.5 yielded far too many false negatives, even after substantial prompt engineering. We had similar results with InstructGPT (`gpt-3.5-turbo-instruct`) and the open-source model StarCoder from HuggingFace. We therefore switched to GPT4 (`gpt-4-0613`).

GPT4, in contrast, initially had a high rate of false *positives*. We noticed that this was partially caused by providing the LLM too much information about the problem, which caused it to fill in details the students had not specified. We therefore removed the problem statement from the prompt; we discuss the consequences of this in section 6. We then performed multiple rounds of prompt revision to bring false positives down to what we felt was an acceptable rate.

Prompt Design. After several iterations, we settled on the following prompt content because it seemed to remove almost all false positives. We analyze the prompt quality in more detail in section 5.3; The full prompt is provided alongside several other supplementary materials in a GitHub repository.[1]

- Examples of any non-primitive data structures (e.g., the Python representation of a "theater hall" (see section 4))
- information about functions which the student has previously defined in another task (including corresponding import statements), if applicable
- the header of the function the student has been asked to define in the current task, but renamed to `student_plan` (so that the function name does not cause GPT4 to generate the correct function body automatically, as sometimes happened)
- the student's plan
- instructions telling the LLM to translate the student's plan step-by-step into equivalent Python code
- strict instructions to **not** correct errors that might be present in the student's plan.

Testing. Once the LLM pipeline produced code, we ran our test suite on the code using the standard Python testing library `unittest` inside Docker instances. Our test suites were sets of up to 5 examples per problem, written by hand.

4 GATHERING STUDENT PLANS

Course context. For this study, our goal was to work with students who knew little programming, so that their programming knowledge would not overly bias their planning. Indeed, we gave them problems that they would *not* be able to program based on their course content to date. Therefore, we deployed the tool in week three of a CS1 course (using the C programming language) at the Graz University of Technology in Austria. At the time of the assignment, students had been introduced to computation in a language-agnostic way by learning about data (types), data storage, abstraction, pseudo-code, and charts.

Student population. The course had a total enrollment of 736 students, most of whom are bilingual. Overall, 544 of the enrolled students submitted a solution for this assignment. In this course, it is not mandatory to submit all assignments to pass them; for this reason, students skip some tasks, especially if the points per exercise do not make up an essential part of the total points. The typical student was was enrolled in a computing-related program such as computer science, software engineering and management,

[1] https://github.com/xstone93/iticse2024-planfeedback/

A plan helps you articulate the high-level structure of your solution to a problem. The plan may include information about different steps that need to be taken, as well as any ordering between different steps. [...]
Example: You want to do your weekly grocery shopping. To get a better structure, you write yourself a list with all the products that you want to buy yourself. Subsequently, you will see two variants of problem plans related to buying groceries. Keep in mind that there are many possible ways to go, you may find easier or more complex plans, which does not mean that one approach might be wrong.
Variant 1:
1. Start with the first element on the list which is not crossed out.
2. Locate the element in the grocery store.
3. Put the desired amount of the element in your shopping bag.
[...]
Variant 2:
1. Categorize items on the list according to similar categories.
2. Start with the first element in the first category, which is not crossed out.
[...]

Figure 2: Planning instructions

or information and computer engineering. Some students had prior programming experience (which we account for in our analysis).

Task Design. We developed three problem scenarios, each with three functions for which students had to write plans. Following our prior work [24], we asked students to write plans as to-do lists. Figure 2 shows (in abbreviated form) the planning instructions given to students (the full instructions are in the GitHub repository). To avoid over-constraining student plans we intentionally showed two distinct plans for the same problem in the instruction document, but students were only asked to produce one.

For seven of the nine functions, students received no feedback; for the other two, they received immediate feedback from the tool. For these two functions, students were allowed to use the tool up to five times.[2] An example of the feedback that students received appears in Figure 3. The spacing of feedback was designed intentionally, as described shortly.

Before students could submit a revised plan for feedback, they had to fill in a single Likert-style question that asked, "How well did you plan do compared to your expectations?" At the end of the assignment, student gave free-response comments on their experience with the tool.

Research Ethics. As this activity was a homework assignment and not intended to be generalizable research, it did not require IRB review in the US. The students were informed before the assignment that the data would be analyzed for research purposes and that their performance or participation had no negative influence

[2]We would have preferred to let students get feedback on every problem and to do so more times. However, due to the cost of using GPT4, even this level of feedback cost over USD 200. Our initial calculation had put the cost even higher, at the limit of our budget, which is why we put these limits in place. We return to this in section 6.

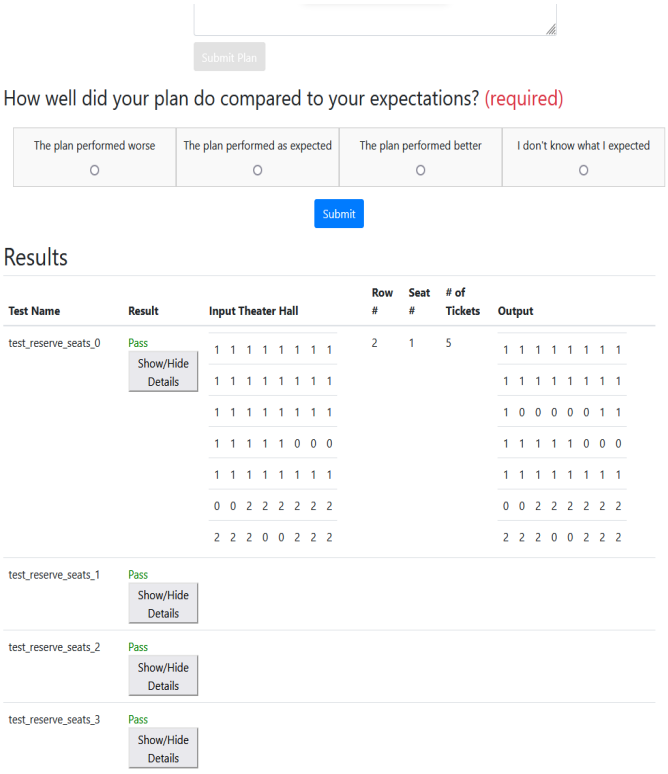

Figure 3: Format of feedback on plans. The example is for the second theater problem. Expanding a row shows the function inputs and the output generated from the synthesized plan.

on their grades (since the data was analyzed anonymously). Furthermore, the students could opt out of the data analysis at any time without disadvantageous consequences. For this reason, IRB was not required in Europe either. We nevertheless took various precautions, such as anonymizing student responses prior to sharing them with the team for analysis.

The Planning Problems. The course instructor (one of the authors) wanted to give problems that reflected the kinds of coding that students would be learning to do later in the course. After reviewing past course assignments and recent planning literature, we settled on three high-level problems, each with three specific functions for which students needed to develop plans. The two functions marked with ‡ are the ones on which feedback was provided.

Theater Seating: Manage reservations in a theater with seats arranged in a 2-dimensional grid.

(1) Count how many seats are available, given information on the current reservations.
(2) ‡ Reserve seats, given information on the current reservations, the row number, the first seat to reserve in that row, and the total number of seats to reserve.
(3) Purchase a ticket, given information on the current reservations, the number of tickets sought, and the ticket category (regular versus premium). The per-ticket price differs based on the category and percentage of seats that are available.

Music Playlists: Create and manage playlists with songs from multiple genres. Information about each song includes title, artist, length, and its genre (e.g., rock, jazz).

(1) Given a playlist and a genre, count how many songs are from that genre.
(2) Calculate total playtime of a genre in a given playlist.
(3) ‡ Check whether the playlist is "balanced" (the number of songs between any two genres differs by at most one).

Restaurant Ordering: Manage a queue of orders.

(1) Add a new order to the end of the queue, given the order details and a reference to the head of the queue.
(2) Remove the order with a given ID from the queue, given a reference to the head of the queue.
(3) Remove the first order from the queue, making the second order the new head of the queue.

Within each problem, the tasks are semi-dependent: a student did not need to solve one before attempting the others, but ideas from earlier tasks might also apply to later tasks.

The feedback problems (‡) were chosen with intent. They are complex enough to require multiple steps in a plan, but we felt they would still be within their reach. In addition, they were intentionally staggered. The second problem was chosen in the first set so the first problem could serve as a "warm-up" activity. Students then went through a few more problems before getting feedback again.

5 ANALYSIS

For our analysis, we wanted to see both the structure *and evolution* of student planning, especially on the functions where students received feedback. It is worth remembering that these are problems for which students (without prior programming experience) *could not yet write programs*.

To make our analysis task tractable, we looked at the planning results on four problems: the three theater-seating problems and the first restaurant problem. We chose the former because they were related but had feedback in the middle. We chose the first restaurant problem because it came after a second round of feedback (on the last music-playlist problem). We therefore assumed impacts from planning feedback would be most salient when contrasting plan styles on the first theater problem and the first restaurant problem.[3]

Given the large number of students, we chose to sample 50 students for analysis. We did not explicitly ask students about their prior programming experience. However, several students mentioned their prior experience, or lack thereof, in their additional free-form responses. We therefore split the students into three groups: those who explicitly mentioned experience (EXP: a total of 34), those who explicitly said they had none (NOV: 25), and everyone else (UNK: the remaining 412). We then chose 10 students each from the prior and no-prior groups, and 30 from the middle group. It is important to note that the middle group will include students both with and without experience, but who did not give any indication of it.

[3]We note that there are many different subsets that could be analyzed; indeed, there is a combinatorial number of them. While there are other reasonable subsets, we feel ours is reasonable enough.

5.1 Plan Iterations

RQ1: To what extent does getting feedback encourage students to revise their plans?

On all the problems with *no* feedback (with one exception below), the median number of submissions per student was 1 and the mean under 1.5. On the two with LLM feedback, the medians were 4 and 2, with means 3.6 and 2.6. This shows that giving feedback made a difference to student plan iteration. The sole exception was the very first problem, which had a median of 2 and mean of 1.9. There, we see that students were still getting used to planning; especially after getting feedback on the second problem, they refined their notion of "a plan" and went back and modified their first plan.

An ANOVA test [15] on the different problems confirmed significant differences across the groups ($p < 0.0001$), and a posthoc Tukey's HSD test [27] confirmed the statistical significance ($p < 0.0001$) of the difference in the means between each of the 3 problems above and all other problems, including each other. In short, students did take advantage of feedback to revise their plans.

5.2 Plan Characteristics

RQ2: What characteristics can we observe in students' plans?

To evaluate the plans, two authors generated four codebooks (which can be found in the GitHub repository). On the first they reached a Cohen κ [6] of 0.8 in 13 rounds. On the second, they reached κ=0.8095 after 10 rounds. On the last two, they reached κ=1 after 8 rounds. For each round, for each of the two functions with feedback (reserve seat and balanced music playlist), five submissions were randomly selected from across all submitted plans.

Table 1 explains the high-level goals and structure of each codebook, which are levels of judgment that go much deeper than what an LLM can reliably provide. As one example, based on these codebooks, the plans in Figure 2 would get codes of RIGHT-TRACK, JUST-RIGHT, PROSE, and LABELED.

We labeled the sampled submissions using these codes. There are several possible dimensions of analysis here. In addition to which problem and which category of student (relative to indicated prior programming experience), each student could also make multiple attempts. For simplicity, we looked at the *first* and *last* submissions students made (even on problems with*out* feedback, students sometimes fixed bugs or revised for other reasons).

The resulting six tables don't fit within this paper (and can be found in full in the GitHub repository). The variation between problems was small enough that the summary of the codes per group, shown in Table 1, is sufficiently informative. Based on this and the detailed data, we observe the following:

Planning Attempts Across all groups, in very few instances do students mostly repeat the problem statement as their "plan" (PROB STATEMENT). Thus, *students are taking the planning process seriously.*

Correctness Students, across all groups and all problems, are on the right track in their first (and last) attempts. Novices and unknown students do a bit worse on Reserve Seats and New Order, but experienced students do not.

Level of Detail About 25% of novices start with extraneous detail, and this number decreases slightly by the end. The experienced and unknown groups have similar rates. Only

Correctness - To avoid relying on the tool's test suites, we instead manually code for "How would a human grader assess this plan?". A plan could be one of: RIGHT TRACK, INCORRECT, PROB-STATEMENT (the plan essentially restated the problem, without decomposing it), or NO ATTEMPT.

	NOV		EXP		UNK	
Code	First	Last	First	Last	First	Last
RIGHT TRACK	32	35	34	35	100	106
INCORRECT	6	4	6	5	13	10
PROB STATEMENT	2	1	0	0	5	3
NO ATTEMPT	0	0	0	0	2	1

Level of Detail - "How much decomposition into tasks did students perform?" One of: JUST RIGHT, NOT ENOUGH, or EXTRANEOUS (the plan included details that were not expected by the problem statement).

	NOV		EXP		UNK	
Code	First	Last	First	Last	First	Last
JUST RIGHT	31	34	29	31	87	91
EXTRANEOUS	9	6	8	7	24	21
NOT ENOUGH	0	0	3	2	6	6

Notation - "What kind of notation did students use?" One of: CODE (mostly using the syntax of a programming language, often C or Python), SEMI-CODE (interleaved programming notation with prose), PROSE, or OTHER.

	NOV		EXP		UNK	
Code	First	Last	First	Last	First	Last
CODE	9	10	8	10	4	5
SEMI CODE	3	3	9	9	11	15
PROSE	28	27	23	21	103	99
OTHER	0	0	0	0	0	0

Structure - Separately, "what kind of visual structure did the plan follow?" One of: CODE (resembled source code) LABELED (ordered sequence of steps), UNLABELED (clear sequence without explicit labeling), PROSE (lacking any visual structure), or OTHER.

	NOV		EXP		UNK	
Code	First	Last	First	Last	First	Last
CODE	10	11	15	17	9	11
LABELED	13	15	21	21	79	79
UNLABELED	7	4	3	1	11	12
PROSE	9	9	0	0	19	17
OTHER	1	1	1	1	0	0

Table 1: Plan Summary Across Problems

the unknown group is large enough to show many instances of insufficient detail, at a rate of about 4%.

Notation Across all three groups, students only rarely change their notation from first to last attempts. The novice group has about three times as many CODE as SEMI CODE, and three times as many PROSE as CODE (so PROSE is about 70% of all submissions). For experienced students, perhaps somewhat surprisingly, the amount of CODE is about the same as for novices, but the amount of SEMI CODE is three times as much (about the same as CODE), so PROSE drops to about 52–57%). The unknown group is curiously different: three times as much SEMI CODE as CODE, and 7–9x as much PROSE as SEMI CODE, so that PROSE is 83–87%. The key takeaway is that in most cases students are writing natural language descriptions (as we had hoped for), not just programs; the rate is (unsurprisingly) much lower for students with programming experience, though some of that could be a consequence of the sample size.

Structure Students did not blindly follow our sample plan structure. Only about 1/3 of novices followed our LABELED structure. Experienced students much more closely followed our sample structure (just over 50%) or otherwise used a code-like structure (just under 50%). The unknown group was much more varied: 66% used LABELED, 10% CODE, 10% UNLABELED, but almost 15% PROSE.

5.3 System Accuracy

RQ3: How accurate is the system in providing feedback?

Our correctness coding also lets us analyze how well the prompt+LLM combination is working. Recall that students get test suite feedback on the generated code. Since we do not expect plans to be complete and perfect, we tolerate some test failures. We thus can refine our definition of false positive to be a case where the plan passes the majority of tests but we give it a code of INCORRECT or NO ATTEMPT, while a false negative fails a majority of tests but we give it a code of RIGHT TRACK.

For false positives, there were only 13 instances of INCORRECT or NO ATTEMPT for the reserve seats problem. 12 out of these 13 plans received system feedback of 0/5 or 1/5 test cases passed. We had 1 outlier, which was a plan that received a 4/5 on the first run, but then the exact same plan received a 0/5 on a future run.

For false negatives, we further break down the results, comparing first-attempt plans (before the first round of feedback) with last-attempt plans. There were 39 plans with a code of RIGHT TRACK on the first attempt. Of these 39, 27 failed a majority of tests. However, by the last attempt, there were 42 plans with a code of RIGHT TRACK, and of these 42, only 19 failed a majority of tests.

In short, the system has an extremely low false positive but moderate false negative rate. Thus, while not very detailed or insightful, testing feedback seems to be a useful proxy for at least the correctness of the plan.

6 DISCUSSION AND LESSONS LEARNED

At a superficial level, we should view this project as a preliminary success. For decades computing education research has asked students to plan, but there has been little ability to *evaluate* plans rapidly enough that students can act on the feedback. The existence of LLMs has made it possible to provide feedback quickly and scalably, enabling students to iterate on their designs.

Another positive takeaway—independent of LLMs—is that students *were able to plan*! As we have seen, they wrote non-trivial plans, instead of just copying the problem text. Furthermore, they could write correct plans for problems they could not yet possibly program. They also iterated their plans in response to feedback.

Another benefit to LLMs is their robustness in the face of human language. They are robust to small typographical and other errors, and can help students who are not native speakers of English.[4]

This paper is not, however, meant to be a celebration of LLM technology. We experienced numerous issues on which (current)

[4]In our case, the language of instruction was English, and the vast majority of plans were in English—but for most students, English was the second or even third language.

LLMs are neutral, unhelpful, or even actively harmful. We spend the rest of this paper discussing these aspects.

Student Opinion. At the end of the assignment, a total of 471 students answered the free-response question: "In what ways did your planning process change when writing the plans with test case feedback? How much did this feedback help/hinder your planning efforts?" We used VADER Sentiment Analysis [16] to identify positive and negative sentiments in their responses, and then topic modeling with Latent Dirichlet Allocation (LDA) [3]. This highlighted three topics for positive and six for negative feedback.
Positive Opinions:

+ **Structure problem planning** - Especially beginners mentioned that the feedback helped to break down and structure ideas, and feedback helped them reflect and refine.
+ **No expressive limitation** - Students with highly detailed plans (that use specific language features) mentioned that they enjoyed having a high level of freedom on what features they use since the tool translates them to code without worrying about it.
+ **Tests consider edge cases** - Beginners noted that test cases were helpful since their initial strategies were too simple and did not consider edge cases.

Negative Opinions:

- **Combatting the LLM** - Some students wondered how to write plans to please the LLM's assessment. They mentioned that they had to restructure their plans to receive better results, even though they were already satisfied with their solution.
- **Attempt counting as a limiting factor** - The limit of five attempts per task made students feel pressured.
- **Testing is not enough** - Students wanted high-level feedback about their plans, not just testing results.
- **Nondeterminism** - Students were frustrated by the LLM's non-determinism, which made tests pass in one run and not the next. Some wanted to see the generated code.[5]
- **Unclear Test cases** - Students mentioned having problems understanding the test cases. They sometimes spent more time understanding the tests than planning.
- **Black box** - Students sometimes did not receive any output due to failure to return values in their plans. This resulted in frustration since it was unclear what to do to pass tests.

What Kind of Feedback? As noted above, the author of a plan primarily wants feedback in terms of the *plan*, not in terms of a distant proxy like a test suite. Unfortunately, this is difficult.

One might think of using an LLM itself for this purpose. In fact, in our early prompts, GPT4—unbidden—provided feedback on the plans, and our eventual prompt had to suppress that. This is because its feedback was (as so often happens with LLMs) sometimes excellent, sometimes unhelpful or a distraction, and sometimes wrong. In the problematic cases, it would give feedback that was wrong or—equally badly—when the plan had an error, just *tell* the student how to fix the plan rather than Socratically leading them to do so for themselves. Of course, it was impossible to tell which case was which. For that reason, we chose to not provide direct LLM feedback.

One intermediate design we used was to have each statement (in a LABELED- or UNLABELED-style plan) turn into a function, converting the plan into a Jupyter notebook. In this setting, with discrete chunks of code, it may be possible to localize flaws (through some form of testing) and provide targeted feedback about specific plan details. However, we did not a priori want to force students into a particular planning style (we offered one as a suggestion, but were curious to see what students would produce). Given that most students do write structured plans anyway, a future version may be able to exploit this structure.

LLMs as Moving Targets. As every reader likely knows, the set of LLMs keeps growing. As noted, even the change from GPT3.5 to GPT4 was significant. We were unable to get sufficiently high-quality output from GPT3.5, which means students would have unreasonably failed tests far too often, while GPT4 has been "too good". In contrast, one of the authors was able to dial down the quality of generated code by asking for it in a less popular language than Python (e.g., Racket).[6] The complex interplay between model, prompt, generated language, and student input makes it difficult to design such tools.

Other Planning Notations. As our prior work [24] has noted, students may also write plans in diagrammatic forms (e.g., as dataflow diagrams). Older LLMs have only processed text. Towards the end of this work, OpenAI released GPT-4V, a model capable of interpreting images as well. We are interested in employing such models so that students have greater flexibility in their planning language.

LLMs Cost Money! While we could have given students effectively unlimited use of GPT3.5 on all the problems, the significantly greater cost of GPT4 (at the time, about 20 times as much) meant we had to significantly limit the number of interactions. While prices did come down during the period of use, the very scale that seems to necessitate LLMs also makes their cost nontrivial. Open-source models have not worked for us, and also require computational resources. While one could imagine training a model specifically to help with planning, that task requires significant expertise.

Including the Right Context. We removed the problem statement from the prompt to prevent the LLM solving the problem irrespective of the plan's details. But this caused its own problems! Students naturally assumed that the problem statement was known to the code synthesizer, and either referenced them or left them out. This caused some otherwise-good plans to fail. This shows that there is a delicate balance to achieving just the right amount of context for an LLM. Since we did not realize this problem until we analyzed the data, we have to leave finding that sweet spot for future work.

Acknowledgements

Thanks to the course staff and students for their participation, and for their patience with this experiment. Work partly supported by NSF awards DRL-2031252 and SHF-2227863.

[5]This would have made little sense in our setting, since many students didn't know how to program at all, and the course was teaching C but the tool generated Python.

[6]Even though the course taught C, we chose to synthesize Python to exploit its testing libraries and to avoid issues like segmentation faults.

REFERENCES

[1] Vincent A.W.M.M. Aleven and Kenneth R. Koedinger. 2002. An effective metacognitive strategy: Learning by doing and explaining with a computer-based cognitive tutor. *Cognitive science* (2002). https://doi.org/10.1207/s15516709cog2602_1

[2] Nicklaus Badyal, Derek Jacoby, and Yvonne Coady. 2023. Intentional Biases in LLM Responses. In *IEEE Ubiquitous Computing, Electronics and Mobile Communication Conference (UEMCON '23)*. https://doi.org/10.1109/UEMCON59035.2023.10316060

[3] David M. Blei, Andrew Y. Ng, and Michael I. Jordan. 2003. Latent Dirichlet Allocation. *Journal of Machine Learning Research* (2003).

[4] Francisco Enrique Vicente Castro and Kathi Fisler. 2016. On the Interplay Between Bottom-Up and Datatype-Driven Program Design. In *ACM Conference on International Computing Education Research (SIGCSE '16)*. https://doi.org/10.1145/2839509.2844574

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

[6] Jacob Cohen. 1960. A Coefficient of Agreement for Nominal Scales. *Educational and Psychological Measurement* (1960).

[7] Michael de Raadt, Richard Watson, and Mark Toleman. 2009. Teaching and Assessing Programming Strategies Explicitly. In *Proceedings of the Eleventh Australasian Conference on Computing Education - Volume 95* (Wellington, New Zealand) *(ACE '09)*. Australian Computer Society, Inc., Darlinghurst, Australia, Australia, 45–54. https://dl.acm.org/doi/10.5555/1862712.1862723

[8] Paul Denny, James Prather, Brett A. Becker, Zachary Albrecht, Dastyni Loksa, and Raymond Pettit. 2019. A Closer Look at Metacognitive Scaffolding: Solving Test Cases Before Programming. In *Koli Calling International Conference on Computing Education Research (Koli Calling '19)*. https://doi.org/10.1145/3364510.3366170

[9] Alireza Ebrahimi. 1994. Novice programmer errors: language constructs and plan composition. *International Journal of Human-Computer Studies* 41 (1994), 457–480. https://doi.org/10.1006/ijhc.1994.1069

[10] James Finnie-Ansley, Paul Denny, Brett A. Becker, Andrew Luxton-Reilly, and James Prather. 2022. The Robots Are Coming: Exploring the Implications of OpenAI Codex on Introductory Programming. In *Australasian Computing Education Conference (ACE '22)*. https://doi.org/10.1145/3511861.3511863

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

[12] Kathi Fisler and Francisco Enrique Vicente Castro. 2017. Sometimes, Rainfall Accumulates: Talk-Alouds with Novice Functional Programmers. In *Proceedings of the 2017 ACM Conference on International Computing Education Research* (Tacoma, Washington, USA) *(ICER '17)*. ACM, New York, NY, USA, 12–20. https://doi.org/10.1145/3105726.3106183

[13] Kathi Fisler, Shriram Krishnamurthi, and Janet Siegmund. 2016. Modernizing Plan-Composition Studies. In *ACM Technical Symposium on Computing Science Education*. https://doi.org/10.1145/2839509.2844556

[14] Luyu Gao, Aman Madaan, Shuyan Zhou, Uri Alon, Pengfei Liu, Yiming Yang, Jamie Callan, and Graham Neubig. 2023. PAL: Program-aided Language Models. In *International Conference on Machine Learning (ICML '23)*. https://proceedings.mlr.press/v202/gao23f.html

[15] Ellen R Girden. 1992. *ANOVA: Repeated measures*. Sage Publications.

[16] C. J. Hutto and Eric Gilbert. 2014. VADER: A Parsimonious Rule-Based Model for Sentiment Analysis of Social Media Text. *Proceedings of the International AAAI Conference on Web and Social Media* 8, 1 (May 2014), 216–225. https://doi.org/10.1609/icwsm.v8i1.14550

[17] Shriram Krishnamurthi and Kathi Fisler. 2021. Developing Behavioral Concepts of Higher-Order Functions. In *ACM Conference on International Computing Education Research*. https://doi.org/10.1145/3446871.3469739

[18] Dastyni Loksa, Lauren Margulieux, Brett A. Becker, Michelle Craig, Paul Denny, Raymond Pettit, and James Prather. 2022. Metacognition and Self-Regulation in Programming Education: Theories and Exemplars of Use. *ACM Transactions on Computing Education* (2022). https://doi.org/10.1145/3487050

[19] O. Muller, B. Haberman, and D. Ginat. 2007. Pattern-oriented instruction and its influence on problem decomposition and solution construction. In *Proceedings of ITiCSE*. ACM, New York, NY, 151–155. https://doi.org/10.1145/1268784.1268830

[20] 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 *"ACM Conference on Innovation and Technology in Computer Science Education - Working Group Reports (ITiCSE-WGR '23)*. https://doi.org/10.1145/3623762.3633499

[21] Brent Reeves, Sami Sarsa, James Prather, Paul Denny, Brett A. Becker, Arto Hellas, Bailey Kimmel, Garrett Powell, and Juho Leinonen. 2023. Evaluating the Performance of Code Generation Models for Solving Parsons Problems With Small Prompt Variations. In *ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE '23)*. https://doi.org/10.1145/3587102.3588805

[22] Robert S. Rist. 1989. Schema Creation in Programming. *Cognitive Science* (1989), 389–414. https://doi.org/10.1016/0364-0213(89)90018-9

[23] Robert S. Rist. 1991. Knowledge Creation and Retrieval in Program Design: A Comparison of Novice and Intermediate Student Programmers. *Hum.-Comput. Interact.* 6, 1 (Mar 1991), 1–46. https://doi.org/10.1207/s15327051hci0601_1

[24] Elijah Rivera, Kathi Fisler, and Shriram Krishnamurthi. 2024. Observations on the Design of Program Planning Notations for Students. 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, 1133–1139. https://doi.org/10.1145/3626252.3630901

[25] Elijah Rivera, Shriram Krishnamurthi, and Robert Goldstone. 2022. Plan Composition Using Higher-Order Functions. In *ACM Conference on International Computing Education Research*. https://doi.org/10.1145/3501385.3543965

[26] James C. Spohrer and Elliot Soloway. 1989. Simulating Student Programmers. In *International Joint Conference on Artificial Intelligence*. 543–549. https://doi.org/doi/abs/10.5555/1623755.1623841

[27] John W. Tukey. 1949. Comparing Individual Means in the Analysis of Variance. *Biometrics* (1949). http://www.jstor.org/stable/3001913

[28] Priyan Vaithilingam, Tianyi Zhang, and Elena L Glassman. 2022. Expectation vs. experience: Evaluating the usability of code generation tools powered by large language models. In *ACM CHI Conference on Human Factors in Computing Systems Extended Abstracts (CHI '22)*. https://doi.org/10.1145/3491101.3519665

[29] Karthik Valmeekam, Matthew Marquez, Alberto Olmo, Sarath Sreedharan, and Subbarao Kambhampati. 2023. PlanBench: An Extensible Benchmark for Evaluating Large Language Models on Planning and Reasoning about Change. arXiv:2206.10498