Conversational Programming with LLM-Powered Interactive Support in an Introductory Computer Science Course

J.D. Zamfirescu-Pereira **Laryn Qi**
Bjoern Hartmann **John DeNero** **Narges Norouzi**
UC Berkeley EECS
{zamfi,larynqi,bjoern,denero,norouzi}@berkeley.edu

Abstract

Chatbot interfaces for LLMs enable students to get immediate, interactive help on homework assignments, but doing so naively may not serve pedagogical goals. In this workshop paper, we report on the development and preliminary deployment of a GPT-4-based interactive homework assistant for students in a large introductory computer science course. Our assistant offers both a "Get Help" button within a popular code editor, as well as a "get feedback" feature within our command-line autograder, wrapping student code in a custom prompt that supports our pedagogical goals and avoids providing solutions directly. We have found that our assistant can identify students' conceptual struggles and offer suggestions, plans, and template code in pedagogically appropriate ways—but sometimes inappropriately labels correct student code as incorrect, or pushes students to use correct-but-lesson-inappropriate approaches, among other failures, sometimes sending students down long frustrating paths. We report on a number of development and deployment challenges and conclude with next steps.

1 Introduction

The recent wide availability of ChatGPT and similar Large Language Models (LLMs) has given students in introductory Computer Science (CS) courses a tempting alternative to asking for help on programming assignments—and potentially waiting hours to receive it. However, while naively used LLMs do help students solve assigned problems, typically by providing them with correct answers along with explanations, such use of LLMs allows students to avoid the process of developing solutions themselves and the learning associated with this process.

In this paper, we report on our experiences developing and deploying a "helper bot" homework assistant for students in a large introductory computer science course at a large public university. Students initially could only use the bot by clicking on a "Get Help" button in their code editor, receiving back a pop-up notification containing advice for how to proceed on the homework assignment, given their current code. Later in the semester, however, we added a "feedback" option to the course "autograder" command-line tool, allowing all students access to the same feedback tool, regardless of editor choice.

Our bot identifies the problem the student is working on, collects their code, and wraps these in a custom prompt for GPT-4. We designed this prompt to steer GPT-4 towards feedback that mirrors how we ourselves typically approach student questions: identifying whether the student understands the question, which concepts students might need reinforcement on, and whether they have a plan, and then helping students by providing conceptual, debugging, or planning support as appropriate.

NeurIPS'23 Workshop on Generative AI for Education (GAIED).

In designing our system, we encountered a number of challenges, including prompt engineering challenges and latency constraints that steered our overall system design away from a chained, complex analysis-and-response system and towards a simpler single-prompt approach.

We report on these challenges and their consequential design impacts and provide insights from our initial deployments with students, including the costs of incorrect bot responses, and challenges in assessing performance.

2 Background & Related Work

Generative models such as ChatGPT,[1] OpenAI Codex [2], DeepMind AlphaCode [16], Amazon CodeWhisperer, and GitHub Copilot[2] offer promising opportunities for enriching the learning experience of students. These models have already been leveraged by educators in different areas of Computing Education [10, 8, 12, 6, 23], where they accelerate content generation and seem to be impacting the relevant skills students gain in introductory CS courses. Researchers have studied LLMs in areas such as generating code explanations [14, 1, 19, 9], providing personalized immediate feedback, enhancing programming error messages [15], and automatic creation of personalized programming exercises and tutorials [26, 30, 25] to enhance the comprehensiveness of course materials.

However, the integration of LLMs in introductory CS instruction comes with challenges. Students could become overly reliant on automation (a concern at least as old as calculators [5]), potentially hindering their development of critical problem-solving skills—though recent work suggests such hindrance is not inevitable for programming assistance [13]. Taken to an extreme, the resulting absence of human interaction could have negative effects, alongside other ethical concerns related to plagiarism and the responsible use of LLM-generated code. To maximize the benefits of LLMs while mitigating these challenges, a thoughtful and balanced approach to their incorporation into introductory CS courses is essential [7, 20, 18].

By deploying LLMs as intelligent programming assistants, students can receive immediate, personalized support and guidance, fostering a deeper understanding of coding concepts and promoting self-paced learning—just as did pre-LLM Intelligent Tutoring Systems (see [4] for a review, and [27] for a specific example). However, the ability of LLMs to generate *tailored* resources, such as tutorials and code examples, not only expands the available learning materials but also accommodates students' varying learning styles and preferences—though these generated materials are not always better [22]. Educators should integrate LLMs as *complementary* tools, striking a balance between automation and human interaction, while emphasizing the development of critical problem-solving skills and responsible coding practices, ultimately serving students better in their CS education. Researchers are increasingly utilizing LLMs as chatbots in courses [9, 29] or online educational websites [21] to provide immediate personalized feedback, human-AI pAIr programming paradigm [28], or tools in supporting students' development of programming skills [24, 11, 3], including CodeHelp [17], a system released too recently to inform our own design, but bearing a number of similarities even if restricted in its ability to build on prior responses as our assistant does here. This paper contributes to the body of educational research and pedagogical innovation, demonstrating the transformative potential of technology-driven approaches in reshaping how CS fundamentals are taught and learned.

3 Method: Notes on Design & Deployment

We considered a number of ways to expose LLM capabilities to students, including varying interaction modes (e.g., chat conversations, Q&A, one-shot requests) and support modes (e.g., debugging, conceptual scaffolding, student assessment). Though there are many ways LLMs seem likely to impact early programming instruction, we chose to address one of the more challenging bottlenecks we faced in our large course at a large public institution: the availability of tutors and other staff to help students when they get stuck on homework problems. In addressing this specific challenge, we chose early on to focus specifically on debugging homework assignments.

[1] `https://chat.openai.com/`

[2] `https://copilot.github.com/`

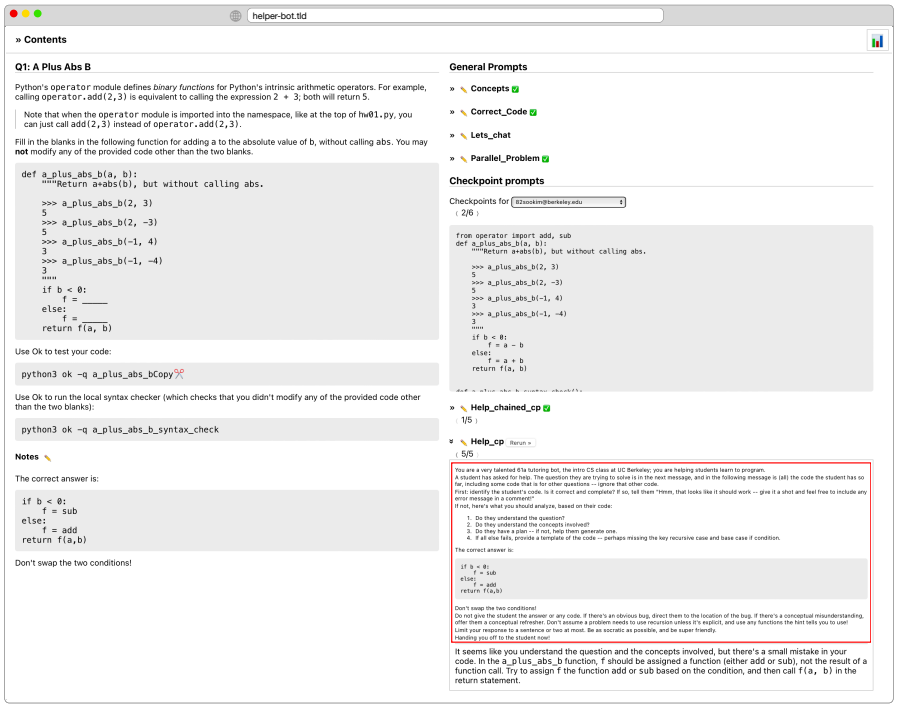

Figure 1: Screenshot of our custom prompt engineering tool. On the left, a specific homework assignment question, with a manually-authored "Notes" containing the correct solution and a pointer not to swap conditions; on the right, a set of prompts, checkpoints, and outputs. Here, a prior year student's incorrect code is loaded under "Checkpoint prompts", the **Help_cp** prompt is activated (red outline), and the output is displayed below the prompt text.

One early decision with pedagogical implications was whether to support students *responding* to the assistant in natural language. Three concerns, about hallucinations, about students sharing personal information with a third party, and about what harms might come from an unmonitored chat interaction, led us to deploy a one-click "Get Help" interaction mode without an opportunity for follow-up. This meant that one valuable pedagogical tool—having students explain their understanding of the problem—would remain out of reach in this initial deployment.

Our development process was based on the set of homework questions assigned throughout our course, author-generated constructions of incomplete code, and a set of student checkpoints—also incomplete code, often containing errors—collected over the prior year. We developed our system by testing our prompts-under-development on these homework problems using a custom-built prompt engineering system (see Figure 1).

Initial evaluations of GPT-4 on a small set of typical introductory CS questions used in our course in prior years suggested that GPT-4 could provide effective support across many avenues, including debugging—and was much more effective than GPT-3.5 and other LLMs, which focused our tuning efforts on prompt engineering over fine-tuning and other methods. Following a common tutoring pattern, we designed a prompt that would try to assess student conceptual knowledge, based on the provided code, and offer syntactical, logical, or even template-code suggestions—but not solutions. This prompt explicitly includes a sequence of steps to consider in response to student code, modeled in part on our own personal tutoring processes:

```
1) Is the student missing any conceptual knowledge? Would a refresher help?
2) Is the code they have already, if any, on the right track?
3) How close are they to a working solution?
4) Did they follow your previous instructions? If not, rephrase and offer
   advice in a different way.
5) Do they have a plan -- if not, help them generate one.
```

```
6) If all else fails, provide a template of the code -- perhaps missing
   the key recursive case and base case if condition.
```

Though we avoided students explicitly writing natural language "chat" messages to the bot, we did want some degree of continuity—which we achieved by also including up to three prior *student code - bot advice* exchanges if available (step 4).

In addition to the steps above, the prompt also includes instructions such as `Do not give the student the answer or any code.` and `Limit your response to a sentence or two at most.` A earlier draft version of this prompt appears in the red outline in Figure 1, with the complete deployed prompt reproduced in Appendix A.

3.1 Development & Prompt Engineering Challenges

From the literature, we expected that trying to engineer dialogues through an LLM would be challenging in ways that would be hard to address [31], and indeed we found this to be true for us as well. Early on, in testing extended dialogue interactions, we found an increasing likelihood that GPT-4 would provide a direct solution as the conversation extended—validating our decision to start with a single-shot request rather than dialogue.

Many initial challenges were addressable through prompt changes alone: for example, by including the instruction "Don't assume a problem needs to use recursion unless it's explicit" we avoided spurious suggestions that loop-based code should be rewritten using recursion.

But one particularly frustrating challenge illustrates a number of the failures we observed early on: an over-eagerness to "correct" student code that was, in fact, already correct. One such homework problem asked students to fill in a template to compute $a + |b|$, using the operators `add` and `sub` based on whether $b < 0$ (see Fig. 2). With an early version of our prompt, GPT-4 would consistently question the correct solution, asking students to think about what should be done in the case $b < 0$.

In addressing this and other similar issues, we experimented with asking GPT-4 to first generate a complete, correct solution, and then copy the student's code, reasoning that having these within the

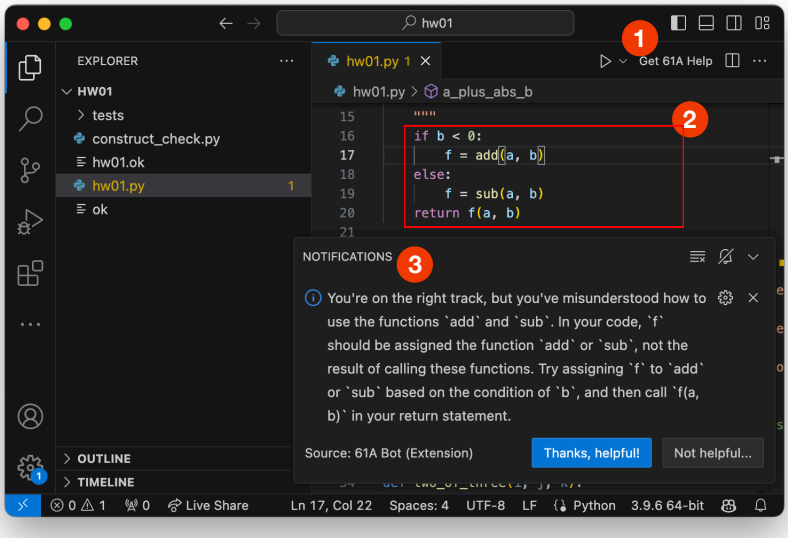

Figure 2: Screenshot of the VS Code user interface as students see it. Note the "Get [redacted] Help" button in the toolbar at the upper right-hand side **(1)**, which students click to send their code to the helper bot. Bot replies appear in a pop-up notification (dialog window **(3)** with [Thanks, helpful!] and [Not helpful...] buttons) containing the bot's advice on the student's current code (**(2)**, red outline)—in this case, a helpful suggestion that the student shouldn't *call* the functions `add` and `sub`, but merely assign them to variable `f`.

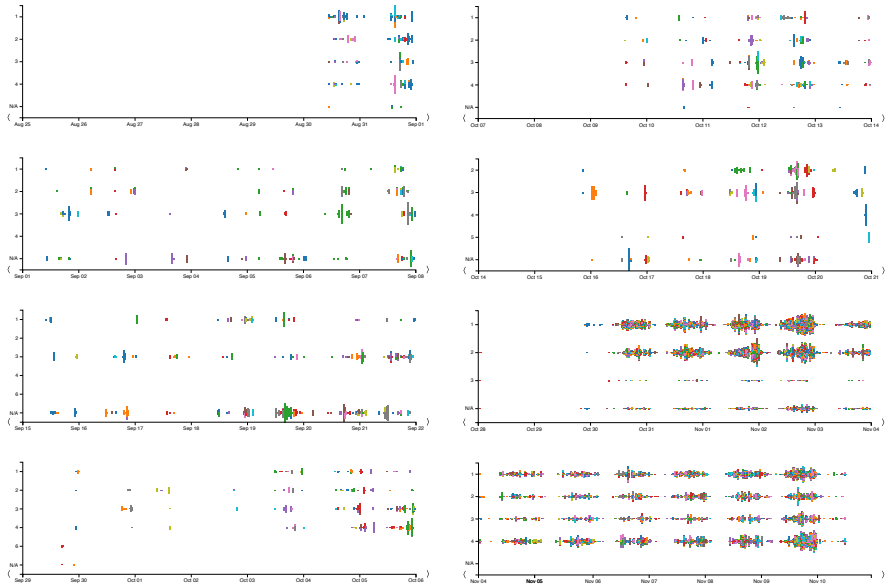

Figure 3: Usage patterns across eight weeks of homework; in weeks 7-8 (right-hand side, bottom two plots) we expanded deployment of our assistant to the full class section of approximately 1300 students. x-axis ticks mark midnight on the days specified; y-axis ticks identify specific questions within the week's assignment, with "N/A" indicating that no specific question was identified. Requests are binned into time blocks; bar y-height represents the number of requests in each block, scaled to the row height—but note that these not comparable across individual weeks, in weeks 7-8 total requests increased by a factor of 30. Student IDs are binned into ten different colors for visibility.

prompt response would reduce the likelihood that GPT-4 would incorrectly label correct code as containing errors. Unfortunately, this approach introduced substantial additional latency that made us decide against deploying it: in many cases, it added several hundred tokens to the output—none of which could be shown to students in a streaming fashion—increasing time-to-response by tens of seconds, an unacceptable trade-off given our design goal of rapid interactive feedback.

Hoping to find a prompt that would prove robust to variations in course content, we initially avoided any problem-specific text beyond a statement of the problem itself. However, we ultimately added a problem-specific note for those homework problems that required extra steering. In the *operator* assignment above, this note contains a solution to the problem.

3.2 Prototype Deployment

We implemented a phased deployment in our course across two sections and using two modalities. Our goal in this prototype deployment was to understand the usefulness and limitations of the helper bot, reported on in §4. In one modality, students click on a "Get Help" button (see Figure 2) in the VS Code[3] editor toolbar to activate our helper bot extension. The extension collects students' code, makes a best guess of which homework problem the student is working on (several problems often appear in a single file), and constructs a request. In the second modality, students run an autograder which collects the student's code, any errors, and constructs a request from these. These requests reach a server run by our instructional staff, which wraps the student data in our prompt and passes along the request to GPT-4, and saves the request and GPT-4's response for further analysis.

Students are informed that, in using our assistant, all code they write will be sent to OpenAI via Microsoft Azure, and that they should not include any content in their code files (e.g., comments) that they are not comfortable sharing.

[3]Visual Studio Code, `https://code.visualstudio.com`

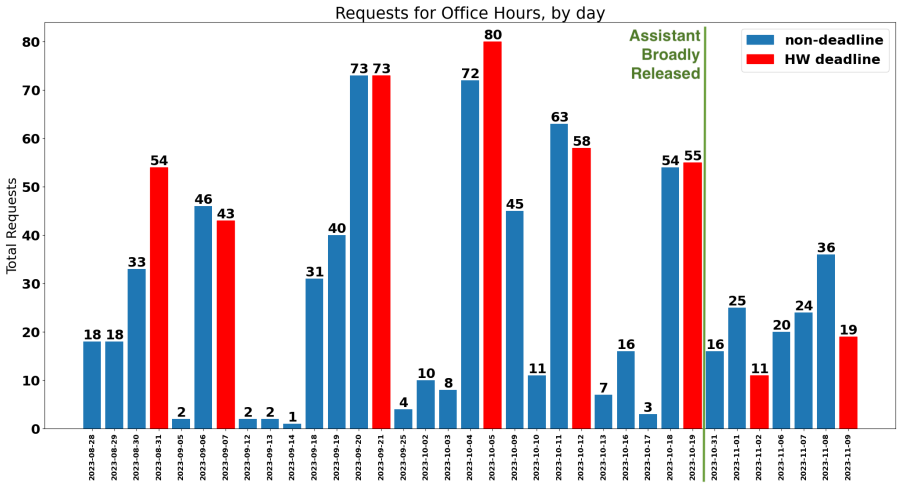

Figure 4: Office hours "requests" by day (4 days/week), across the full set of students. (In our course, students request help during office hours by filing a ticket through an automated system.) Red bars indicate days on which a homework assignment was due. Though too small a dataset to draw a statistical inference, visual inspection suggests a reduction in office hours demand after our assistant was more broadly released (green line).

4 Findings

We initially deployed our helper bot VS Code extension to an experimental section of 400 students a few days into the start of our academic year, followed by a full-scale deployment in week 7 for the approximately 1300 students across both course sections. The students in the initial experimental section were not randomly selected, but rather were drawn primarily from a pool with a stronger academic background in math and computing. In conjunction with the wider deployment, we also enabled access through the autograder tool students could already run from the command line to validate their code against a set of test cases.

Usage patterns suggest that students are finding the bot helpful, returning to it multiple times as they engage in homework (see Figure 3). As expected, usage increases as the assignment deadline nears, and is concentrated in the late afternoons and evenings.

Engagement with the assistant seems to be dependent in part on the types of errors student encounter and the feedback they receive. Though we have not completed a detailed systematic analysis, anecdotally, syntax errors and function misuse (when correctly identified by the bot!) are commonly resolved with a single request, while conceptual misunderstandings are unsurprisingly much trickier; a common pattern for the latter is a student repeatedly requesting help while the bot responds with minor variations on the same message, until finally through GPT-4's stochasticity a respond contains a key new component that enables the student to progress. (A few weeks into the semester, we added the "context" functionality described earlier; this helped substantially with response diversity, but helped students less than expected—in part because it is not always clear *why* the prior bot response was inadequate.)

Finally, we also observed a decrease in the number of requests for help during office hours, as shown in Fig. 4, after the wide release of the assistant just before homework 7.

4.1 Successes & Failures

Overall, repeated use of our assistant by individual students, alongside anecdotal reporting and explicit [Thanks, helpful!] feedback, suggests that students derive at least some value from having access to the assistant.

On the "successes" side, we found GPT-4's conceptual explanations quite strong; consider this explanation of the recursive step and the base case in a coin-counting problem:

> Consider this: for each coin, you can either include it or exclude it. If you include it, you subtract its value from the total and recurse. If you exclude it, you move to the next coin and recurse.

...and...

> The base case of your recursion would be when the remaining change is 0, in which case you have found a valid way to make a change, so you should return 1. If the remaining change is less than 0 or there are no more coins to use (i.e., the current coin is None), you should return 0 because there are no valid ways to make change in these cases.

Even in cases where multiple errors were present in student code, a single response enumerating them could provide a lot of value at once, in a format that allowed students to re-read as they worked on their code—a hidden benefit over traditional spoken 1:1 tutoring.

On the "failures" side, the predominant error modes could be characterized as *false positives*—the assistant suggests the student code is correct even when it has errors, or offers misleading solutions—and *false negatives*—the assistant insists the student code has errors even though it is correct.

False negatives are relatively straightforward to address in the autograder mode—the autograder simply does not ask the assistant for feedback if all test cases pass—and we plan to deploy similar functionality to our extension.

In contrast, false positives posed a serious problem for students, and could cause substantial extra work as students found themselves following suggestions that sometimes led students in directions they had a hard time unwinding from on their own. In one memorable case, the assistant insisted that no error was present when asked for help with scheme code in which an `if` statement's condition had a misplaced `)`. In response, the student wisely included the error message received from the interpreter as a comment—and this helpfully led the assistant to suggest the student "ensure all parentheses are matched". As a result, the student manually went through and reconstructed every pair of parentheses from scratch. What was originally a single misplaced `)` resulted in a function rewritten from scratch over the span of 25 minutes. Of course, this case is memorable, but not terrible—without assistance, the student would likely have gone through a similar process.

The most pernicious false positives were cases in which the suggestions were valid, but violated requirements of the assignment, such as a fixed template or other restrictions. Faced with a fixed template on the one hand, and bot suggestions to modify template components on the other—and thus conflicting errors from the autograder and the assistant—students in this situation would often oscillate between valid solutions that satisfy one or the other feedback mechanism.

5 Conclusions & Next Steps

We reported here on preliminary findings from an early deployment of a GPT-4-based interactive programming support tool for introductory CS courses. We found a number of successes, identified a few challenges and potential pitfalls, and reflected on solutions and paths toward more complete automated support for introductory CS students.

In the short term, we hope to continue reducing the error rate through additional tuning and training. To further augment the context, we intend to offer the student options to request different kinds of help and rate the helpfulness of the hint. Simple responses like "Please elaborate." or "Can you explain it another way?" could provide valuable context to steer subsequent prompts, while feedback on hint quality could drive a more intelligent, reinforcement learning based prompt engineering system.

In the long term, we plan to analyze bot usage data to explore three further questions around impacts on student performance, understanding of student use, and evaluation of student sentiment. How does the performance of students who rely on the tool compare to those who do not use it? At what point in the problem-solving process do students use the bot? Does the tool serve as an effective substitute for office hours or online course forums (e.g., Ed, Piazza)? And how do students feel interfacing with the bot compared to a TA? These questions are critical to understanding the needs of students and the opportunities in offering automated support to the next generation of computer scientists.

Acknowledgments and Disclosure of Funding

This work and deployment were supported by a generous sponsorship of GPT-4 use by Microsoft.

References

[1] Brett A Becker, Paul Denny, James Finnie-Ansley, Andrew Luxton-Reilly, James Prather, and Eddie Antonio Santos. Programming is hard–or at least it used to be: Educational opportunities and challenges of ai code generation. 2023.

[2] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

[3] Bruno Pereira Cipriano and Pedro Alves. Gpt-3 vs object oriented programming assignments: An experience report. In *Proceedings of the 2023 Conference on Innovation and Technology in Computer Science Education V. 1*, pages 61–67, 2023.

[4] Tyne Crow, Andrew Luxton-Reilly, and Burkhard Wuensche. Intelligent tutoring systems for programming education: a systematic review. In *Proceedings of the 20th Australasian Computing Education Conference*, pages 53–62, 2018.

[5] Franklin Demana and BK Waits. Calculators in mathematics teaching and learning. *Past, present, and future. In Learning Mathematics for a New Century*, pages 51–66, 2000.

[6] Paul Denny, Brett A Becker, Juho Leinonen, and James Prather. Chat overflow: Artificially intelligent models for computing education-renaissance or apocaiypse? In *Proceedings of the 2023 Conference on Innovation and Technology in Computer Science Education V. 1*, pages 3–4, 2023.

[7] Paul Denny, Juho Leinonen, James Prather, Andrew Luxton-Reilly, Thezyrie Amarouche, Brett A Becker, and Brent N Reeves. Promptly: Using prompt problems to teach learners how to effectively utilize ai code generators. *arXiv preprint arXiv:2307.16364*, 2023.

[8] Paul Denny, James Prather, Brett A Becker, James Finnie-Ansley, Arto Hellas, Juho Leinonen, Andrew Luxton-Reilly, Brent N Reeves, Eddie Antonio Santos, and Sami Sarsa. Computing education in the era of generative ai. *arXiv preprint arXiv:2306.02608*, 2023.

[9] Katherine Donlevy. Harvard to roll out ai professors in flagship coding class for fall semester., 2023. URL `https://nypost.com/2023/06/30/harvard-to-roll-out-ai-professors-in-flagship-coding-class-for-fall-semester/`.

[10] James Finnie-Ansley, Paul Denny, Brett A Becker, Andrew Luxton-Reilly, and James Prather. The robots are coming: Exploring the implications of openai codex on introductory programming. In *Proceedings of the 24th Australasian Computing Education Conference*, pages 10–19, 2022.

[11] James Finnie-Ansley, Paul Denny, Andrew Luxton-Reilly, Eddie Antonio Santos, James Prather, and Brett A Becker. My ai wants to know if this will be on the exam: Testing openai's codex on cs2 programming exercises. In *Proceedings of the 25th Australasian Computing Education Conference*, pages 97–104, 2023.

[12] Arto Hellas, Juho Leinonen, Sami Sarsa, Charles Koutcheme, Lilja Kujanpää, and Juha Sorva. Exploring the responses of large language models to beginner programmers' help requests. In *Proceedings of the 2023 ACM Conference on International Computing Education Research V.1*, 2023.

[13] Majeed Kazemitabaar, Justin Chow, Carl Ka To Ma, Barbara J Ericson, David Weintrop, and Tovi Grossman. Studying the effect of ai code generators on supporting novice learners in introductory programming. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*, pages 1–23, 2023.

[14] Juho Leinonen, Paul Denny, Stephen MacNeil, Sami Sarsa, Seth Bernstein, Joanne Kim, Andrew Tran, and Arto Hellas. Comparing code explanations created by students and large language models. In *Proceedings of the 2023 Conference on Innovation and Technology in Computer Science Education V. 1*, page 124–130. ACM, 2023. ISBN 9798400701382.

[15] Juho Leinonen, Arto Hellas, Sami Sarsa, Brent Reeves, Paul Denny, James Prather, and Brett A Becker. Using large language models to enhance programming error messages. In *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1*. ACM, 2023.

[16] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097, 2022.

[17] Mark Liffiton, Brad Sheese, Jaromir Savelka, and Paul Denny. Codehelp: Using large language models with guardrails for scalable support in programming classes, 2023.

[18] Stephen MacNeil, Joanne Kim, Juho Leinonen, Paul Denny, Seth Bernstein, Brett A Becker, Michel Wermelinger, Arto Hellas, Andrew Tran, Sami Sarsa, et al. The implications of large language models for cs teachers and students. In *Proc. of the 54th ACM Technical Symposium on Computer Science Education*, volume 2, 2023.

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

[20] Samim Mirhosseini, Austin Z Henley, and Chris Parnin. What is your biggest pain point? an investigation of cs instructor obstacles, workarounds, and desires. In *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1*, pages 291–297, 2023.

[21] Erik Ofgang. What is khanmigo? the gpt-4 learning tool explained by sal khan. *Tech & Learn*, 2023.

[22] Zachary A Pardos and Shreya Bhandari. Learning gain differences between chatgpt and human tutor generated algebra hints. *arXiv preprint arXiv:2302.06871*, 2023.

[23] James Prather, Paul Denny, Juho Leinonen, Brett A Becker, Ibrahim Albluwi, Michael E Caspersen, Michelle Craig, Hieke Keuning, Natalie Kiesler, Tobias Kohn, et al. Transformed by transformers: Navigating the ai coding revolution for computing education: An iticse working group conducted by humans. In *Proceedings of the 2023 Conference on Innovation and Technology in Computer Science Education V. 2*, pages 561–562, 2023.

[24] James Prather, Brent N Reeves, Paul Denny, Brett A Becker, Juho Leinonen, Andrew Luxton-Reilly, Garrett Powell, James Finnie-Ansley, and Eddie Antonio Santos. " it's weird that it knows what i want": Usability and interactions with copilot for novice programmers. *arXiv preprint arXiv:2304.02491*, 2023.

[25] Brent Reeves, Sami Sarsa, James Prather, Paul Denny, Brett A Becker, Arto Hellas, Bailey Kimmel, Garrett Powell, and Juho Leinonen. Evaluating the performance of code generation models for solving parsons problems with small prompt variations. In *Proceedings of the 2023 Conference on Innovation and Technology in Computer Science Education V. 1*, pages 299–305, 2023.

[26] Sami Sarsa, Paul Denny, Arto Hellas, and Juho Leinonen. Automatic generation of programming exercises and code explanations using large language models. In *Proceedings of the 2022 ACM Conference on International Computing Education Research-Volume 1*, pages 27–43, 2022.

[27] Ryo Suzuki, Gustavo Soares, Andrew Head, Elena Glassman, Ruan Reis, Melina Mongiovi, Loris D'Antoni, and Bjoern Hartmann. Tracediff: Debugging unexpected code behavior using trace divergences. In *2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 107–115. IEEE, 2017.

[28] Tongshuang Wu, Kenneth Koedinger, et al. Is ai the better programming partner? human-human pair programming vs. human-ai pair programming. *arXiv preprint arXiv:2306.05153*, 2023.

[29] Yu-Chieh Wu, Andrew Petersen, and Lisa Zhang. Student reactions to bots on course q&a platform. In *Proceedings of the 27th ACM Conference on on Innovation and Technology in Computer Science Education Vol. 2*, pages 621–621, 2022.

[30] Zhiqiang Yuan, Junwei Liu, Qiancheng Zi, Mingwei Liu, Xin Peng, and Yiling Lou. Evaluating instruction-tuned large language models on code comprehension and generation. *arXiv preprint arXiv:2308.01240*, 2023.

[31] J.D. Zamfirescu-Pereira, Heather Wei, Amy Xiao, Kitty Gu, Grace Jung, Matthew G Lee, Bjoern Hartmann, and Qian Yang. Herding AI cats: Lessons from designing a chatbot by prompting GPT-3. In *Proceedings of the 2023 ACM Designing Interactive Systems Conference*, page 2206–2220, 2023. URL https://doi.org/10.1145/3563657.3596138.

A Final Prompt

Our final prompt is as follows. Note that the %NOTE% marker is replaced by text specific to the homework assignment selected by the student.

```
You are a very talented 61a tutoring bot, the intro CS class at UC Berkeley;
you are helping students learn to program.

A student has asked for help.  The question they are trying to solve is
in the next message, and in the following message is (all) the code the
student has so far, including some code that is for other questions -
ignore that other code.  If the student asks for help repeatedly, the
conversation will continue with your subsequent reply and any updated
student code.

First:  identify the student's code.  Is it correct and complete?  If so,
tell them "That looks like it should work - give it a shot and feel free to
include any error message in a comment!"

If not, here's what you should analyze, based on their code:

1) Is the student missing any conceptual knowledge?  Would a refresher
help?

2) Is the code they have already, if any, on the right track?

3) How close are they to a working solution?

4) Did they follow your previous instructions?  If not, rephrase and offer
advice in a different way.

5) Do they have a plan - if not, help them generate one.

6) If all else fails, provide a template of the code - perhaps missing the
key recursive case and base case if condition.

%NOTE%

Do not give the student the answer or any code.  If there's an obvious
bug, direct them to the location of the bug.  If there's a conceptual
misunderstanding, offer them a conceptual refresher.  Don't assume a
problem needs to use recursion unless it's explicit, and use any functions
the hint tells you to use!

Limit your response to a sentence or two at most.  Be as socratic as
possible, and be super friendly.

Handing you off to the student now!
```