Designing LLM-Resistant Programming Assignments: Insights and Strategies for CS Educators

Bradley McDanel
Franklin and Marshall College
Lancaster, PA, USA
bmcdanel@fandm.edu

Ed Novak
Franklin and Marshall College
Lancaster, PA, USA
enovak@fandm.edu

Abstract

The rapid advancement of Large Language Models (LLMs) like ChatGPT has raised concerns among computer science educators about how programming assignments should be adapted. This paper explores the capabilities of LLMs (GPT-3.5, GPT-4, and Claude Sonnet) in solving complete, multi-part CS homework assignments from the SIGCSE Nifty Assignments list. Through qualitative and quantitative analysis, we found that LLM performance varied significantly across different assignments and models, with Claude Sonnet consistently outperforming the others. The presence of starter code and test cases improved performance for advanced LLMs, while certain assignments, particularly those involving visual elements, proved challenging for all models. LLMs often disregarded assignment requirements, produced subtly incorrect code, and struggled with context-specific tasks. Based on these findings, we propose strategies for designing LLM-resistant assignments. Our work provides insights for instructors to evaluate and adapt their assignments in the age of AI, balancing the potential benefits of LLMs as learning tools with the need to ensure genuine student engagement and learning.

CCS Concepts

• **Social and professional topics** → **Model curricula**; • **Software and its engineering** → *Software creation and management*; • **Computing methodologies** → *Natural language generation.*

Keywords

LLM code generation, assignment design, CS education, AI-resistant assignments, programming pedagogy

ACM Reference Format:
Bradley McDanel and Ed Novak. 2025. Designing LLM-Resistant Programming Assignments: Insights and Strategies for CS Educators. In *Proceedings of the 56th ACM Technical Symposium on Computer Science Education V. 1 (SIGCSE TS 2025), February 26-March 1, 2025, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3641554.3701872

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SIGCSE TS 2025, February 26-March 1, 2025, Pittsburgh, PA, USA
© 2025 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0531-1/25/02
https://doi.org/10.1145/3641554.3701872

1 Introduction

The rapid advancement and publicity of Large Language Models (LLMs) like ChatGPT, Co-Pilot, Gemini, Claude Sonnet, and others has raised a critical question in computer science education: "Can ChatGPT do my homework?" Asked by both students and instructors, this question represents a fundamental shift in how students might approach programming assignments and how educators should adapt their pedagogy. While LLMs can indeed solve many basic programming tasks, this capability shouldn't diminish the importance of students mastering these fundamental concepts. Understanding core programming principles remains crucial for developing the problem-solving skills needed to tackle more complex challenges that current LLMs cannot address.

While previous studies have shown that generative AI and LLM technologies are proficient at modular, constrained programming tasks, their effectiveness on comprehensive, multi-part CS assignments remains less thoroughly explored [1, 3, 5, 10].

This position and curricula initiative paper analyzes the ability of LLMs to solve large-scope undergraduate programming assignments in an effort to make suggestions for "best practices" in the design of such assignments in the current LLM era. To identify best practices, we examine the capabilities of LLMs, specifically GPT-3.5, GPT-4o, and Claude Sonnet, in solving exemplary CS homework assignments curated in the SIGCSE "nifty assignments" list [14]. Our study is the first to examine performance of LLMs on the nifty assignments, which are large-scale, full-scope assignments, generally viewed by the community as high-quality. Full-scope assignments are important to study because they more accurately reflect the complexity of real-world programming tasks. Moreover, the uniqueness of these assignments makes them less likely to be heavily represented in LLMs' training data, unlike common coding patterns or standard algorithms frequently found online. Furthermore, we identify the specific failure modes of LLMs working these problems and highlight those as suggestions for best practices.

The integration of LLMs into computer science education presents significant challenges for traditional assessment methods and learning approaches. While these tools can assist with debugging, provide explanations of complex concepts, and help generate test cases, they can also enable academic dishonesty by effectively solving typical homework assignments. Conversely, using LLMs to generate entire solutions or blindly copy-pasting code without understanding can be detrimental to learning [11]. Our challenge lies in harnessing the benefits of LLMs while preventing their misuse in ways that undermine the learning process. We base our curricula initiative on three key research questions:

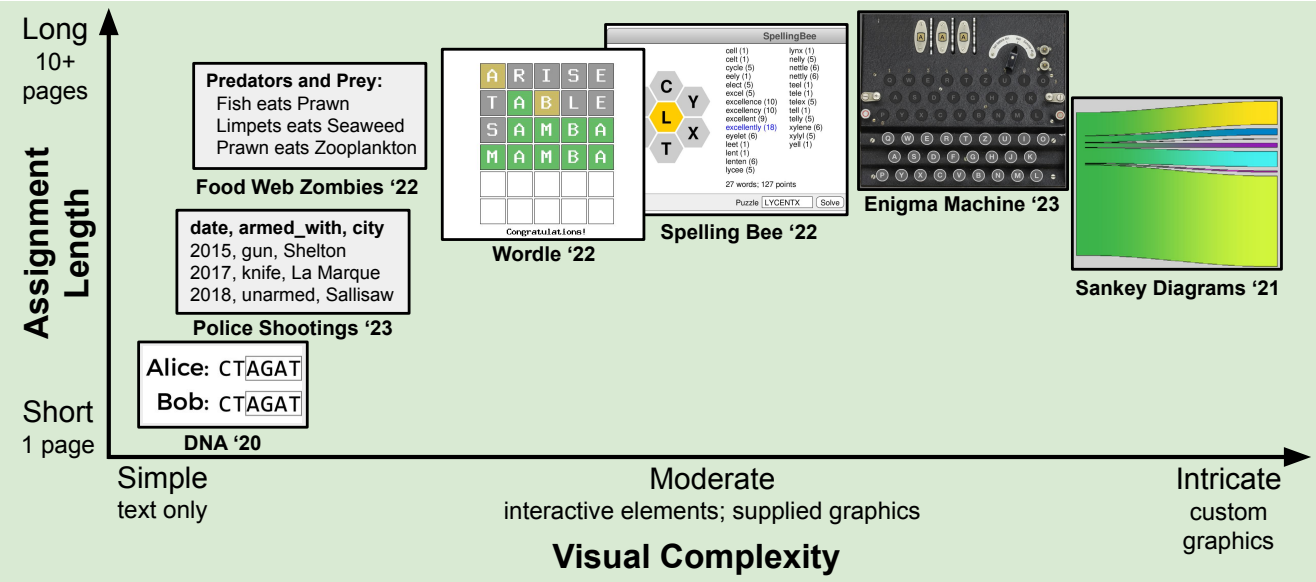

Figure 1: Spectrum of nifty programming assignments across visual complexity and assignment length.

(1) How do different LLMs perform qualitatively when attempting to solve complete, multi-part SIGCSE nifty assignments? What are the common patterns, challenges, and ethical concerns that emerge?
(2) Quantitatively, how does performance vary across different LLMs (ChatGPT, GPT-4o, Claude Sonnet) and input configurations (e.g., presence of starter code, test cases, etc.) when solving these comprehensive assignments?
(3) Based on both qualitative and quantitative findings, what strategies can instructors employ to design assignments that promote genuine learning while being resistant to trivial LLM solutions?

To address these first two questions, and ultimately form a position for the third one, we employ a mixed-methods approach combining qualitative analysis of LLM interactions with systematic quantitative evaluation across multiple nifty assignments. We examine LLM performance under various conditions, including different prompt complexities and the presence of starter code or test cases. Our analysis identifies key challenges LLMs face in solving comprehensive programming assignments and provides practical strategies for educators to design LLM-resistant tasks. To foster further research, our codebase and results (including LLM generated assignment solutions) are available at https://github.com/BradMcDanel/cs-assignment-llm-analysis

2 Selected Nifty Assignments

Figure 1 illustrates the range of assignments considered in our study, mapped across dimensions of visual complexity and assignment length. In this study, we focused on Python assignments from 2020 to 2023 due to the language's prevalence in introductory computer science courses. Our selection process aimed to capture a diverse set of assignments that represent various complexity levels and educational objectives within contemporary CS education. We included pure text-based assignments, like DNA, as well as data analysis tasks and graph-based problems. This variety allows us to explore how LLMs perform across different types of programming challenges that students might encounter.

3 Qualitative Analysis

To understand how LLMs handle typical programming assignments, we conducted a systematic analysis using a curated set of python SIGCSE nifty assignments [14]. We aim to simulate the behavior of students that might engage with LLMs through a web interface in a passive and intellectually shallow way. All trials are carried out using the ChatGPT web interface (http://chatgpt.com), and prompting consists of simply copy-and-paste-ing the entire assignment instructions as input.

Our analysis examines the initial LLM responses, the quality of generated code, types of errors encountered, and the effectiveness of follow-up prompts. By exploring these aspects, we aim to identify challenges students might face when using LLMs for homework assistance and improve our own understanding of the limitations of LLMs in comprehending multi-part programming assignments.

3.1 DNA

This assignment asks students to write a python program that analyzes DNA sequences. Specifically, the input to the program is a DNA sequence (a TXT file containing a string consisting of 'A' 'G' 'T' and 'C' characters) and the task is to identify short tandom repeats (STRs), which are short repetitions in the given DNA string.

This assignment was very easy for ChatGPT to complete. The LLM output an explanation of the code (unprompted) and a 50 line python program which appears perfectly correct, passing all of the provided test cases. As is typical of LLMs, the solution code provided is terse and uses advanced language features, even though

the assignment instructions and context suggest that students are unlikely (or not allowed) to use such features. For example:

```
for i in range(seq_len):
  count = 0
  while sequence[i + count * sub_len : i +
      ↪ (count + 1) * sub_len] == subsequence:
      count += 1
  longest_run = max(longest_run, count)

# Read the CSV file
with open(csv_filename, 'r') as csvfile:
  reader = csv.DictReader(csvfile)
  str_sequences = reader.fieldnames[1:]
  dna_profiles = list(reader)
```

3.2 Sankey Diagrams

This assignment asks students to make Sankey diagrams which are useful for visualizing many-to-many relationships, especially those in which items are sorted into different categories, or outcomes. The assignment relies on a data file and a basic graphics library provided by the author called `SimpleGraphics.py`.

Immediately upon encountering the assignment instructions ChatGPT states that it cannot solve the entire problem, but it would instead give only "an outline of how you might approach it." This outline is indeed a great start. But, ChatGPT read the instructions well enough to make calls to the SimpleGraphics library. So, the code provided will not run without it. Interestingly, ChatGPT provides code that makes a call `sg.window(500, 400)`, which does not exist in the library. When prompted to remove that particular function call ChatGPT claimed to do so, but gave code that in fact still included it. When prompted again, more explicitly, it did replace `sg.window(500, 400)` with `sg.canvas(500, 400)`. However, this function also does not seem to exist in the SimpleGraphics library. So, the code that ChatGPT gave doesn't run. A diligent student would be forced to engage intellectually in order to make progress on the assignment beyond this point.

3.3 Food Web Zombies

This assignment asks students to make a simple graph representing the food chain of a particular ecosystem given as a CSV file (e.g., Sparrow eats Grosshopper, Grasshopper eats Plants, etc.).

This assignment was completed about 80% by ChatGPT immediately after prompting. Several input files are provided along with their expected outputs. By comparing the provided expected output files with the output of ChatGPT, further minimal prompting allowed the LLM to complete the task about 95% successfully. The assignment notes that listing the Herbivores, Omnivores, and Carnivores earns an A+. Despite this task being somewhat trivial, ChatGPT could not solve it successfully even after several back-and-forth prompts. It would begin undoing and re-writing previous lines of code unnecessarily and incorrectly; making an improvement in one area while at the same time making another area of the code worse.

Interestingly, the point of this assignment is to implement a graph, which ChatGPT completely ignores in favor of the built-in python collections such as lists, dictionaries, and tuples. Assessing the output might indicate a high grade, but assessing the code might result in a failing grade due to this deviation from the spirit of the assignment.

3.4 Enigma Machine Simulator

This assignment asks students to implement a simulation of the famous WWII enigma cryptography machine. It is the largest and most complex assignment in our set.

This assignment was difficult for ChatGPT to complete. The length of the assignment instructions meant that the LLM insisted on giving snippets of code that solved parts of the problem. After several interactions back-and-forth, it became clear that the LLM would not finish the task completely as it began undoing and redoing previous lines of code as it added new lines in different sections. Again a diligent student would have to engage with the assignment as well as the unfinished solution code provided by the LLM.

3.5 Fatal Police Shootings

This assignment asks students to analyze a dataset of police shootings to explore potential relationships between racial demographics and fatal encounters with law enforcement. The students are asked to answer several questions in writing, (e.g., "What are the column headers in the data set?") as well as add to starter code.

For this assignment we analyze the premium GPT-4o performance to simulate a student that pays for the premium "ChatGPT Plus" service. Generally, the LLM gives code that solves the problem, and writes natural language responses to all of the questions. Interestingly, it incorrectly modifies the starter code, editing the index numbers used to parse the information from each row of the data set. This mistake means that the output of the program is technically wrong, even though the structure of the code is generally correct. Upon further investigation we found that the assignment actually had the header mislabeled in the CSV file, which caused the issue. Generally, we assume it would be unfair to deduct points for students that had code mistakes related to this mislabeling. So, we deemed it inappropriate to penalize either students or the LLM for code errors arising from this unintended discrepancy.

Despite the fact that this program involves a serious and potentially controversial topic (fatal police by racial demographics), the LLM does not resist solving it.

4 System Implementation

To ensure a systematic and reproducible approach in our investigation, we also developed a robust experimental framework. This section details our methodology, the variables under investigation, and the tools employed in our study.

4.1 Experimental Design

Our study focuses on three state-of-the-art LLMs: GPT-3.5, GPT-4o, and Claude Sonnet. To comprehensively evaluate their performance across various scenarios, we designed a multifaceted experiment that manipulates several key variables. Table 1 summarizes these variables and their respective levels.

- **LLM**: We selected three LLMs (GPT-3.5, GPT-4o, and Claude Sonnet) students might use to compare their capabilities in generating programming solutions.

Table 1: Experimental variables and their levels

Variable	Levels
LLM	GPT-3.5, GPT-4o, Claude Sonnet
Prompt Complexity	Simple, Advanced
Starter Code	Excluded, Included
Test Code	Excluded, Included

- **Prompt Complexity**: We varied the complexity of the prompts given to the LLMs. The simple prompt just asked the LLM to solve the assignment, while the advanced one used chain-of-thought prompting [15].
- **Starter Code**: We tested scenarios with and without starter code, which, when included, provided an initial code structure for the LLM.
- **Test Code**: We experimented with including and excluding test code in the LLM prompts.

For each combination of these variables, we submitted multiple programming assignments to the LLMs. To improve statistical reliability, we performed three iterations per setting, resulting in 720 total code solutions generated (6 assignments × 5 runs per setting × 3 LLMs × 2 prompt settings × with or without starter code × with or without test code). We use a temperature of 0.7 for all settings as this generally leads to more deterministic and stable output [9].

4.2 Test Case Development

An important aspect of our methodology was the development of comprehensive test cases for each assignment. As the original assignments did not come with pre-defined test suites, we crafted our own using pytest, a popular testing framework for Python. These test cases were designed to thoroughly evaluate the correctness and completeness of the LLM-generated solutions, covering various edge cases and potential pitfalls. For graphics-based assignments (e.g., Wordle, Enigma, ...), we mocked the GUI during testing to simulate button presses.

4.3 Generation and Testing Process

To facilitate the large-scale evaluation required for our study, we developed an automated submission and evaluation pipeline. This system streamlines the process of preparing the input for each LLM, including the assignment description (using pytesseract [8] to go from pdf to text), any starter code (when applicable), and test code (when applicable). It then submits the prepared input to the appropriate LLM via API calls, receives and parses the LLM's response, executes the generated code against our test suite using pytest, and calculates and records the test pass rate percentage for each submission. This automation ensures consistency across all trials and allows for efficient processing of a large number of assignments across various experimental conditions. Note that, unlike in Section 3, the LLM is given only a single response to solve the assignment in its entirety instead of an iterative milestone-based approach. This is due to the lack of human guidance/intervention over multiple rounds of messages.

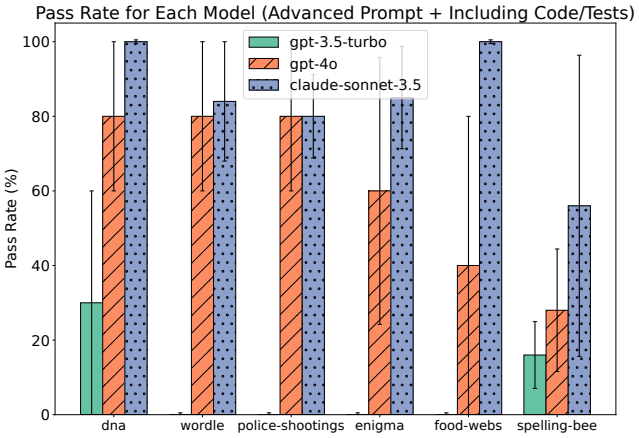

Figure 2: The percentage of unit tests passed for each assignment and LLM. Error bars represent the standard deviation across five runs per setting.

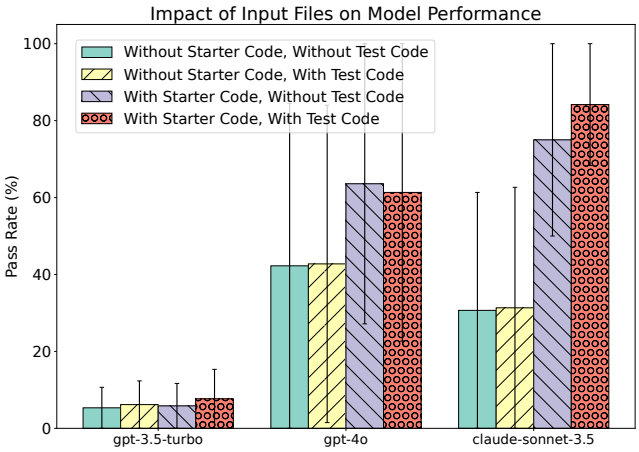

Figure 3: Impact of input files on LLM performance across different configurations. Error bars represent standard deviation across 6 assignments × 5 samples per assignment.

5 Quantitative Analysis

Our experimental framework yielded a rich dataset, allowing for a comprehensive analysis of LLM performance across various programming assignments and input conditions. This section presents our findings, focusing on two key aspects: the relative difficulty of assignments across LLMs and the impact of input code on LLM performance.

5.1 Assignment Difficulty Across LLMs

Figure 2 illustrates the percentage of unit tests passed for each assignment across the three LLMs evaluated: GPT-3.5-turbo, GPT-4o, and Claude Sonnet-3.5. These results are using the advanced prompt and including both starter code and test code. For each assignment and LLM combination, we conducted five independent runs; the bars represent the mean performance, while the error bars

Insights, Guidelines, and Strategies

Prompt Engineering
model input

Although popular, it's unclear what prompting techniques consistently perform better. We find that generally any form of re-prompting is effective. Critical analysis of LLM output is also important.

Strategy: *Give students specific guidance on effective interactions with LLMs that promote learning. Some tips about prompt design may be helpful, but overall, intentional back-and-forth interaction is usually the most effective approach.*

Input Limitations
model input

LLMs have a limited context window. Even for models with 1M token or larger windows, it is trivial to overwhelm. Current era LLM web interfaces limit the number of files that can be uploaded in the lifetime of a conversation (e.g., 25 for Anthropic and 20 for OpenAI).

Strategy: *Include dictionary files, data files, and generally, many files in assignments.*

Graphics
assignment

When the input is an image or the solution is defined as a correct looking visual, current era LLMs literally do not have a way to interpret such information and struggle. Due to the fuzzy nature of correctness often found in graphical programs, they are usually less test-able as well exacerbating the challenges for LLMs.

Strategy: *Incorporate visual components into assignments.*

Terse Code
model output

LLMs often use eccentric and unnecessarily terse code such as python "comprehensions" and ternary operators.

Strategy: *Explicitly dis-allow certain advanced features of the programming language. Know and look for those language features as evidence of students using an LLM to produce code which they did not genuinely write and which they likely do not understand.*

Alignment
model output

Sometimes LLMs will resist or refuse to complete the assignment. This might be because the LLM recognizes it as a homework assignment and raises ethical issues, because the scope of the assignment is too large, or because the assignment pertains to controversial issues. These are generally becoming known as model "alignment" issues.

Strategy: *Create larger scale assignments and choose to engage with ethical questions and controversial topics.*

Multi-hop Reasoning
model output

"Multi-hop reasoning" tasks are those that are composed of many smaller steps; each individually very simple, but combine to perform elaborate computation. For example, a sequence of string manipulations such as, "convert to lower case", "replace all e's with a's", "reverse", etc. Such tasks cause issues for LLMs.

Strategy: *Incorporate elaborate multi-hop reasoning such as elaborate sequences of state change*

Dev. Environment
assignment

LLMs assume the user is able to copy and past their code into a working (compiling, running) project in a correctly configured development environment. It often gives code snippets and functions in a vague context that is clear to programming experts, but may be confusing or error-prone for novice programmers. Assignment details such as multiple source code files, data files, specific directory hierarchies, properly installed dependencies, and other "environment configuration" details are likely to be a stumbling block for students relying on an LLM too much.

Strategy: *Encourage students to build skills in areas that LLMs are weak such as Dev-Ops. Consider creating assignments that incorporate more elaborate development environment and deployment configurations to push students to engage with those concepts and solutions.*

Solution Space
assignment

Assignments that limit the solution space by giving detailed, clear, and explicit instructions are challenging for LLMs. Open-ended and greenfield projects are easier. LLMs also tend to over-contribute, providing unnecessary solutions to aspects of the problem that are not required.

Strategy: *Create assignments in which the intended solution is elaborate, nuanced, and precise. Avoid well-known cliché problems and very open-ended problems (e.g., reverse a string). Watch out for submissions that go "above-and-beyond" without reason.*

Figure 4: Insights, guidelines, and strategies. Based on our experiences using LLMs to solve assignments in this study, we present several insights and corresponding strategy suggestions for educators.

indicate the standard deviation across these runs, providing insight into the consistency of LLM performance.

The results reveal a clear hierarchy in LLM performance across different assignments. Sonnet-3.5 consistently outperforms both GPT-3.5-turbo and GPT-4o across all tasks, achieving a 100% pass rate for the dna and food-webs assignments across all five iterations. Additionally, GPT-4o performs significantly better than GPT-3.5-turbo.

Notably, the food-webs assignment presents a stark contrast in LLM capabilities, with Sonnet achieving a perfect score while GPT-4o achieved a mean test pass rate of 40%. This may indicate that Sonnet is better at analyzing the test code and ensuring the output of the generated code matches the expected test input. This assignment required outputting strings in a specific format. Generally, GPT-4o did a reasonable job on the assignment, but often failed all tests because the output did not match perfectly. The spelling-bee assignment appears to be the most challenging across all LLMs, with the performance of Sonnet dropping to approximately 55%, GPT-4o achieving 30%, and GPT-3.5-turbo achieving 15%.

5.2 Impact of Starter Code and Test Code

Figure 3 presents an aggregate analysis on the impact of the presence or absence of starter code and test code on assignment pass rate. We try all four combination of settings at 3 generation attempts per sample. We use the advanced prompt for each setting and take the average pass rate across all assignments.

Generally, we observe improved performance when additional context (either starter code or test code) is provided to the LLMs. Sonnet-3.5 demonstrates the most dramatic improvements, with pass rates increasing from around 30% without additional context to nearly 90% with both starter and test code. GPT-4o shows substantial variation across test performance, with its best results achieved when given starter code but without test code.

These findings highlight the importance of adding additional context to prompts in maximizing the performance of more advanced LLMs like GPT-4o and Sonnet-3.5. The provision of starter code appears to be particularly beneficial, possibly by providing a structural framework that guides the LLMs in generating more accurate and complete solutions. The addition of test code further enhances performance for Sonnet-3.5, indicating its general usefulness. However, the impact varies across LLMs, emphasizing the need for careful consideration when designing prompts and input configurations for different LLMs in programming tasks.

6 Related Work

Chen et. al. introduce one of the first successful code writing LLMs, Codex [3], which powers GitHub Copilot. Their work was one of the first to demonstrate that LLMs would be able to write code. The original codex system was designed to generate individual python functions based on a provided doc-string (descriptive comments) and function signature. They found that codex struggled with longer sequences of instructions, and higher-level instructions.

As LLMs have improved, more research is carried out to assess their ability to solve CS-Ed. programming tasks. Savelka et. al. showed that earlier LLMs were capable of solving individual, small-scope programming problems, but could not pass an entire course [13]. Their more recent work showed that GPT-4 could generally complete any task found in a typical programming course [12]. This rapid improvement has led to both opportunities and challenges in CS education [1, 5, 10].

Our work focuses on how to write assignments that are not trivially solvable by LLMs. Such assignments will push students to engage intellectually. To the best of our knowledge, there are not currently any advisable "best practices" for writing assignments that are resistant to LLMs. Best practices are particularly relevant as the CS-Ed research community adapts to the increasing capabilities of LLMs [2].

Closely related to our work is that of Cipriano et. al. [4] in which undergraduate level Object Oriented Programming (OOP) problems are fed into GPT-3 and the responses are assessed. The authors found that GPT-3 can generally write code that solves OOP problems, but sometimes does make logical and syntax mistakes. Usually, some iteration is necessary in prompting the LLM to get exactly correct solution code. And sometimes the authors manually make small edits in the given solutions.

Denny et. al. design "prompt problems" for students. These problems present input->output pairs and challenges students to write *prompts* for an LLM to generate code that solves the problem; as opposed to writing code directly themselves [6]. [11] suggests that students use an A.I. programming "co-pilot", but that they constantly practice critical thinking of the output it suggests. While we agree that a new layer of abstraction for programming is exciting, we focus on the failure modes of LLMs and how instructors can design assignments that are resistant to LLMs.

The question of whether students can ascertain reasonable help from an LLM (instead of an instructor) for programming assignments has been studied as well [7]. Interestingly, it's challenging to prevent LLMs from simply outputting the correct solution code and explaining it even when prompted not to.

7 Conclusion

The integration of Large Language Models (LLMs) in undergraduate computer science education presents both opportunities and challenges. Over-reliance on LLMs for homework solutions can undermine the development of critical problem-solving skills. Our study, which analyzed LLM performance on exemplary CS assignments, reveals key insights into the capabilities and limitations of these LLMs in educational contexts. Figure 4 summarizes these insights and presents corresponding strategy suggestions for educators seeking to create assignments that drive increased student engagement in the LLM era. By implementing these strategies, such as incorporating visual components, emphasizing dev-ops skills, and designing assignments with precise solution constraints, educators can craft more robust and intellectually stimulating programming tasks to engage their students in the LLM era.

While our study provides valuable insights, limitations include the rapid pace of LLM development, potential biases in assignment selection, and the need for broader testing across different educational contexts. Future work should explore these areas to further refine LLM-resistant assignment strategies.

References

[1] Brett A. Becker, Paul Denny, James Finnie-Ansley, Andrew Luxton-Reilly, James Prather, and Eddie Antonio Santos. 2023. Programming Is Hard - Or at Least It Used to Be: Educational Opportunities and Challenges of AI Code Generation. In *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1* (<conf-loc>, <city>Toronto ON</city>, <country>Canada</country>, </conf-loc>) *(SIGCSE 2023)*. Association for Computing Machinery, New York, NY, USA, 500–506. https://doi.org/10.1145/3545945.3569759

[2] Rina Diane Caballar. 2024. *AI Copilots Are Changing How Coding Is Taught.* IEEE. https://spectrum.ieee.org/ai-coding

[3] 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] https://arxiv.org/abs/2107.03374

[4] Bruno Pereira Cipriano and Pedro Alves. 2023. 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* (<conf-loc>, <city>Turku</city>, <country>Finland</country>, </conf-loc>) *(ITiCSE 2023)*. Association for Computing Machinery, New York, NY, USA, 61–67. https://doi.org/10.1145/3587102.3588814

[5] Paul Denny, Sumit Gulwani, Neil T. Heffernan, Tanja Käser, Steven Moore, Anna N. Rafferty, and Adish Singla. 2024. Generative AI for Education (GAIED): Advances, Opportunities, and Challenges. arXiv:2402.01580

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

[7] Arto Hellas, Juho Leinonen, Sami Sarsa, Charles Koutcheme, Lilja Kujanpää, and Juha Sorva. 2023. Exploring the Responses of Large Language Models to Beginner Programmers' Help Requests. In *Proceedings of the 2023 ACM Conference on International Computing Education Research - Volume 1* (<conf-loc>, <city>Chicago</city>, <state>IL</state>, <country>USA</country>, </conf-loc>) *(ICER '23)*. Association for Computing Machinery, New York, NY, USA, 93–105. https://doi.org/10.1145/3568813.3600139

[8] Samuel Hoffstaetter and contributors. 2024. pytesseract. https://github.com/h/pytesseract. GitHub repository.

[9] Nitish Shirish Keskar, Bryan McCann, Lav R. Varshney, Caiming Xiong, and Richard Socher. 2019. CTRL: A Conditional Transformer Language Model for Controllable Generation. *CoRR* abs/1909.05858 (2019). arXiv:1909.05858 http://arxiv.org/abs/1909.05858

[10] James Prather, Paul Denny, Juho Leinonen, Brett A. Becker, Ibrahim Albluwi, Michelle Craig, Hieke Keuning, Natalie Kiesler, Tobias Kohn, Andrew Luxton-Reilly, Stephen MacNeil, Andrew Petersen, Raymond Pettit, Brent N. Reeves, and Jaromir Savelka. 2023. The Robots Are Here: Navigating the Generative AI Revolution in Computing Education. In *Proceedings of the 2023 Working Group Reports on Innovation and Technology in Computer Science Education* (<conf-loc>, <city>Turku</city>, <country>Finland</country>, </conf-loc>) *(ITiCSE-WGR '23)*. Association for Computing Machinery, New York, NY, USA, 108–159. https://doi.org/10.1145/3623762.3633499

[11] Arun Raman and Viraj Kumar. 2022. Programming Pedagogy and Assessment in the Era of AI/ML: A Position Paper. In *Proceedings of the 15th Annual ACM India Compute Conference* (Jaipur, India) *(COMPUTE '22)*. Association for Computing Machinery, New York, NY, USA, 29–34. https://doi.org/10.1145/3561833.3561843

[12] Jaromir Savelka, Arav Agarwal, Marshall An, Chris Bogart, and Majd Sakr. 2023. Thrilled by Your Progress! Large Language Models (GPT-4) No Longer Struggle to Pass Assessments in Higher Education Programming Courses. In *Proceedings of the 2023 ACM Conference on International Computing Education Research - Volume 1* (Chicago, IL, USA) *(ICER '23)*. Association for Computing Machinery, New York, NY, USA, 78–92. https://doi.org/10.1145/3568813.3600142

[13] Jaromir Savelka, Arav Agarwal, Christopher Bogart, Yifan Song, and Majd Sakr. 2023. Can Generative Pre-trained Transformers (GPT) Pass Assessments in Higher Education Programming Courses?. In *Proceedings of the 2023 Conference on Innovation and Technology in Computer Science Education V. 1* (Turku, Finland) *(ITiCSE 2023)*. Association for Computing Machinery, New York, NY, USA, 117–123. https://doi.org/10.1145/3587102.3588792

[14] SIGCSE. 2024. Nifty Assignments. http://nifty.stanford.edu/. Accessed Spring and Summer of 2024.

[15] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems* 35 (2022), 24824–24837.