StudentEval: A Benchmark of Student-Written Prompts for Large Language Models of Code

Hannah McLean Babe
Oberlin College
Oberlin, OH, USA

Sydney Nguyen
Wellesley College
Wellesley, MA, USA

Yangtian Zi
Northeastern University
Boston, MA, USA

Arjun Guha
Northeastern University and Roblox
Boston, MA, USA

Molly Q Feldman
Oberlin College
Oberlin, OH, USA

Carolyn Jane Anderson
Wellesley College
Wellesley, MA, USA

Abstract

Code LLMs have the potential to make it easier for non-experts to understand and write code. However, current CodeLLM benchmarks rely on a single expert-written prompt per problem, making it hard to generalize their success to non-expert users. In this paper, we present a new natural-language-to-code benchmark of prompts written by a key population of non-experts: beginning programmers. StudentEval contains 1,749 prompts written by 80 students who have only completed one introductory Python course. StudentEval contains numerous non-expert prompts describing the same problem, enabling exploration of key factors in prompt success. We use StudentEval to evaluate 12 Code LLMs and find that StudentEval is a better discriminator of model performance than existing benchmarks. Our analysis of student prompting strategies reveals that nondeterministic LLM sampling can mislead students about the quality of their descriptions, a finding with key implications for Code LLMs in education.

1 Introduction

Large language models trained on both natural language and programming language text (Code LLMs) have the potential to democratize programming: less-experienced programmers can interact with models in natural language to write, edit, and explain code. A growing body of work looks at their potential impact on the productivity of professional programmers [5, 25, 27], and a number of evaluation benchmarks have been developed for the natural language-to-code task [3, 8, 13, 15, 16]. However, the evaluation of Code LLMs has for the most part been modeled after experts, both in the choice of test problems, and by having professional programmers write the natural language prompts.

Popular benchmarks such as HumanEval [8] and MBPP [3] consist of many problems from varying areas of computing, accompanied by a single expert-written prompt. Achieving good performance on these benchmarks indicates that a model will perform well across many programming tasks, *assuming that the user can write prompts equally as well as the expert.*

In this paper, we present a Code LLM benchmark that asks how well models perform when users do not know how to sound like experts. Our StudentEval dataset contains 1,749 written by beginning CS students, validated with expert-written test cases, and constructed using a novel approach that sets it apart from prior work in three key ways. **1)** Existing benchmarks have prompts authored by more experienced programmers, whereas StudentEval has *prompts authored by students who have only completed one computer science course.* **2)** Existing benchmarks contain tricky problems designed to stress-test the problem solving capabilities of Code LLMs. In contrast, StudentEval has problems that are *easily solved with expert-written descriptions, but often fail with student-written descriptions.* **3)** Existing benchmarks only have a single prompt per problem, whereas *StudentEval has on average 36 prompts per problem, representing a variety of prompting skill levels.* This diversity lets us explore what it means to write a "good" prompt.

The StudentEval problems target a specific skill level and provide a diverse set of prompts for each problem along with expert-written test cases. Students wrote English descriptions of these problems in an iterative manner in collaboration with a Codex model. Prompts were collecting using 48 beginner-appropriate problems; StudentEval contains numerous different prompts for each problem. Our prompts exhibit the variation in technical vocabulary and lack of familiarity with how to describe code that are common with non-experts. While other researchers have considered novice student interactions with Code LLMs [17], StudentEval is the first benchmark based on student interactions. This framing provides significant insight into Code LLM reasoning capabilities outside of the educational context.

Our key contributions are:

- We present StudentEval, a benchmark consisting of 1,749 student-written descriptions of natural-language-to-code tasks.
- Using four key subsets of the StudentEval benchmark, consisting of descriptions that pass (fail) on the first (last) attempt by a student, we evaluate 12 state-of-the-art Code LLMs. Our results show that StudentEval is better able to discriminate between models than the popular HumanEval benchmark.
- We conduct an in-depth analysis of the prompts and find that even successful student prompts lead models to generate multiple semantically distinct programs.

2 Background

Existing benchmarks pair natural language descriptions of code with test cases to check the validity of generated programs. The two most commonly used benchmarks, HumanEval [8] and MBPP [3], are in Python. There are also multi-language benchmarks that translate problems from one language to another [2, 6]. Finally, there are alternate benchmark formats, including multi-turn evaluation [22] and docstring generation [20].

General-purpose benchmarks Most existing benchmarks have a single natural language description of a problem, typically written by an expert programmer. There are a few exceptions that scrape the web or crowdsource [1, 13, 16], but expert-written benchmark predominate. These benchmarks provide wide coverage, but come with limitations. First, they have a *single* prompt per problem. Consider this HumanEval prompt:

Imagine a road that's a perfectly straight infinitely long line. n cars are driving left to right; simultaneously, a different set of n cars are driving right to left. The two sets of cars start out being very far from each other. All cars move in the same speed. Two cars are said to collide when a car that's moving left to right hits a car that's moving right to left. However, the cars are infinitely sturdy and strong; as a result, they continue moving in their trajectory as if they did not collide. This function outputs the number of such collisions.

While the correct solution is simply n^2, the prompt is designed to be confusing. Models succeed or fail based on this *specific phrasing*. Having a single prompt precludes explorations of how crucial word choice, grammar, etc. is to model success. StudentEval's non-expert, multi-prompt construction enables us to analyze what makes a successful prompt: each problem has at least 14 prompts that describe the task in a different way.

Second, existing benchmarks contain problems at widely varying difficulty levels. Compare the problem above, which requires mathematical reasoning that may challenge many programmers, with a trivial problem from the same benchmark [8]: *Return length of given string.* Although these benchmarks cover a wide range of programming tasks, it is difficult to interpret their results as evidence that a model will or won't suit a particular group of programmers, since they aggregate over very different skill levels.

Domain-specific benchmarks There are also a handful of domain-specific benchmarks, such as DS-1000 [16] and MathQA [3]. Like these domain-specific benchmarks, we target a specific population of programmers; however, we target a particular skill level rather than an application area. In addition, we provide numerous non-expert prompts per problem.

Beyond natural language to code This paper and the aforementioned work focuses on the natural language to code task. However, there are a number of benchmarks that focus on a variety of other tasks, including program repair, code editing, test generation, and more (e.g., [7, 9, 21, 23]).

3 The StudentEval Dataset

In this section we describe StudentEval, a many-prompt-per-problem benchmark that targets a specific programmer skill level.[1] The dataset consists of 1,749 English-language prompts for 48 programming problems, with at least 14 prompts per problem. All prompts were written by university students who had completed a single semester of computer science in Python (CS1). These students represent a population of programmers with a uniform knowledge base, allowing us to choose problems that they all should be able to solve. The data was collected in Spring 2023, during the first six months that ChatGPT was available. However, our discussions with participants indicate that participants were generally unfamiliar with Generative AI.

Problem Selection and Format Given our goal of collecting many non-expert descriptions for each problem, we compiled a suite of 48 programs that closely resembled the kinds of problems that are familiar to students. The majority of problems were pulled directly from CS1 course materials, with light modifications to avoid publishing answers to assignments still in use. Thus, all participants should be able to understand and solve the problems by directly writing solutions in Python; we explore whether they are also able to describe them in natural language so that Code LLMs can solve them. The problems exercise a variety of Python features. For topic diversity, we define 8 core concepts: lists, loops, strings, conditionals, math, nested data, sorting, and dictionaries.

Each StudentEval problem consists of three components: a function signature, a reference implementation, and 3+ test cases that achieve high coverage on the reference implementation (Figure 1). When we gather student data, which we describe below, we show participants only a function's signature and test cases. From this information, they produce a description, which we automatically validate using the problem's test cases.

Problem Validation We validated our problems in several ways. For common problems (e.g. factorial), LLMs can produce working implementations from the function name alone. To weed out these problems, we produced Codex [8] generations from each function signature with no docstring and measured mean pass@1 rate [8], which is a standard metric that estimates the probability that the Code LLM produces a solution that passes all tests in one shot, calculated over 200 samples. Overall, the mean pass@1 for our signatures without docstrings is 0.0519 with a variance of 0.0364. The maximum pass@1 is 0.925, for the problem exp.

We also validated the test suites associated with each problem. The test cases serve two roles in our dataset collection: they help students understand the problem, and they ensure that the LLM-generated solutions are correct. Liu et al. [19] give evidence that the test cases that accompany widely-used Code LLM benchmarks frequently miss important corner cases. To avoid this pitfall, we use both test coverage and mutation testing of the reference implementation to ensure that the test cases in StudentEval are adequate. Unlike in Liu et al. [19], the StudentEval tests need to be understood by students who have only completed CS1. We strive for a balance between exhaustiveness and comprehensibility: each problem has 3–4 tests that achieve 100% code coverage. Mutation testing [14] is a more rigorous way than coverage to measure the quality of a test suite, and we used MutPy [12] to compute mutation scores. All mutation scores below 90 are either the result of MutPy generating no mutations or generating a technically correct mutation that still passes tests.

Gathering 1,749 Student-Written Prompts We recruited 80 beginning CS students from three U.S. higher education institutions to build the StudentEval benchmark. The study was IRB-approved - we obtained consent prior to and verbal assent during the study. We conducted the study over Zoom, using a web-based application designed specifically for StudentEval . This application presents the function signature and tests for one problem at a time. Students

[1]Available at https://huggingface.co/datasets/wellesley-easel/StudentEval

Function signature (visible)	`def convert(lst):`

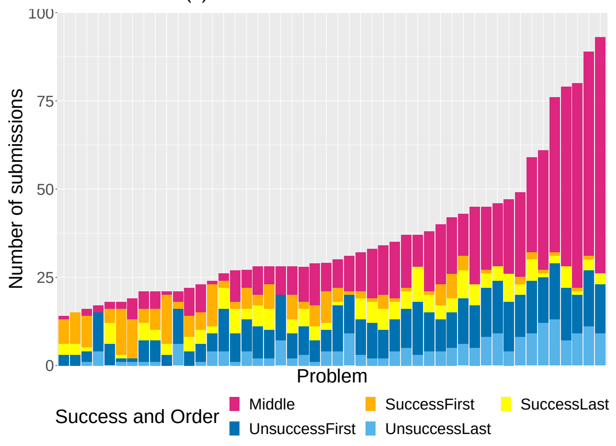
Expert tests (visible to student; hidden from model; automatically run on generated code)	**Input** / **Expected Output**
	`[0,1,2,3]` → `['ABCD']`
	`[0,-1,1,-1,2]` → `['A','B','C']`
	`[1,1,1,-1,25,25,-1,0,1,2]` → `['BBB','ZZ','ABC']`

Student description (pass@1 = 0.8)	*takes a list of numbers. Create a ABC list with the capital letters in the alphabet and create an answer string. Iterate through the input list, if there is "-1" then add ' ' to the answer string, or otherwise, add the letter with the corresponding index of the answer string. Split the answer string at ' '. return the answer string.*
Student description (pass@1 = 0.0)	*Assign a number from 0~25 to each alphabet, and create a list of string of alphabetical letters based on their assigned numbers in the lst. When there is -1 in the lst, create a new string and add it to the list. Return a list of created strings.*

Figure 1: An example StudentEval problem. Our web-based experiment platform shows students the signature and expert-written tests. When students submit their description, we use a Code LLM to generate code, test it, and flag failed tests for the students. StudentEval has numerous student-written descriptions of each problem.

Subset	Items	Word Count
First Failure	450	28.8 (25.5) ± 16.7
First Success	187	28.8 (25.0) ± 17.4
Last Failure	205	35.9 (30.0) ± 22.6
Last Success	185	37.8 (35.0) ± 18.4

(a) Sizes and word counts.



(b) Attempts per problem.

Figure 2: The four subsets of StudentEval.

enter a problem description into a text box. After they submit, our server constructs a prompt with the function signature and their problem description formatted as a Python docstring, and sends this prompt to Codex to produce the function body. The server then tests the function in a sandbox and presents the test results to the participant. Students had the option to reattempt the problem or move on to the next problem. Participants completed three tutorial and 8 StudentEval problems in 75 minutes, receiving a $50 gift card for participation.

Dataset Subsets and Basic Statistics Students generated 1,749 prompts in total, with an average of 36 prompts per problem. There are significant variations in how the prompts differ from each other: many are small, iterative changes (+/- a few words) whereas a student's first, last, and successful prompts tend to vary significantly from others. To refine the dataset for evaluation, we break the StudentEval dataset into four disjoint subsets (Figure 2a): students most frequently failed to solve problems on their first attempt, and this is the largest subset of problems (*First Failure*); about half as many first attempts were successful (*First Success*); slightly fewer students gave up after multiple attempts (*Last Failure*); and others succeeded after multiple attempts (*Last Success*). In cases where a student attempts a problem exactly once, we classify it as a First Failure or First Success. These subsets omit "Middle" prompts (Figure 2b), which are failures and neither first nor last attempts. Figure 2a shows that *Last* descriptions are significantly longer than *First*, which suggests students add detail even when starting afresh might be better.

4 Results

We evaluate 12 Code LLMs. We focus our comparison on gpt-3.5-turbo, the three "Python specialist" Code Llama models [4], the four StarCoderBase models [18], and Phi-1 [11]. We confirm that none of the StudentEval prompts appear in The Stack, the open training dataset for StarCoderBase and other models. As with other benchmarks, we use hidden unit tests to evaluate the correctness of model-generated code. We use standard generation parameters for all models.[2]

4.1 How Do Models Perform on StudentEval?

Table 1 reports the mean pass@1 rate for every model on the four subsets of StudentEval. We include HumanEval pass@1 rates for comparison.

Code Llama models perform best We find that *the Code Llama models significantly outperform all other models on the* First/Last

[2]We use temperature 0.2, 0.95 top-p sampling, and generate up to 512 new tokens.

Table 1: Mean pass@1 for the models that we evaluate on the four subsets of StudentEval.

Model (Size)	First Failure	Last Failure	First Success	Last Success	HumanEval
GPT-3.5-Turbo-0301 (?)	10.86	**12.41**	44.84	47.40	48.1
Phi-1 (1.3B)	11.28	8.37	59.16	36.36	51.22
StarCoderBase-1B (1B)	1.77	1.21	24.86	13.00	15.17
StarCoderBase-3B (3B)	5.91	5.66	51.73	32.20	21.46
StarCoderBase-7B (7B)	5.49	6.82	62.35	46.42	28.37
StarCoderBase (15.5B)	7.82	6.74	65.28	51.74	30.40
Code-Llama-Py-7B (7B)	6.51	8.59	66.88	55.36	40.48
Code-Llama-Py-13B (13B)	9.56	9.33	70.22	62.26	42.89
Code-Llama-Py-34B (34B)	**11.40**	10.14	**73.51**	**64.65**	**53.29**

Success *prompts.* The 13B model outperforms StarCoderBase-15B, the closest competing model, by 5-10% (absolute). The 34B model performs even better.

StudentEval exposes a bigger gap between large and small models than HumanEval HumanEval is the de facto standard code benchmark; and many Code LLM developers strive to obtain good HumanEval scores, and many smaller LLMs have achieved HumanEval scores that are competitive with larger LLMs. However, we observe that the difference between pass@1 rates for large and small models is more substantial with StudentEval than HumanEval. 1) For the StarCoderBase models, pass@1 on *Last Success* is almost 4x higher with the 15B model vs the 1B model, but the gap is much smaller (2x) on HumanEval. 2) Phi-1 (1.3B) approaches Code Llama (34B) on HumanEval, but Code Llama is 1.7x better on *Last Success* than Phi-1. Phi-1's HumanEval performance comes from its "textbook quality" training data. Unfortunately, novices don't write textbook quality prompts. Our results suggest that natural data is better at helping non-experts.

4.2 Variation in Pass@1

Most Code LLM papers only report mean pass@1 for a benchmark, averaging over problems with widely varying pass rates. Because StudentEval contains multiple prompts per problem, it illuminates the extent to which luck plays a role in whether a Code LLM produces the right answer for a user. In Figure 3, we group prompts by problem, so the plots show the percentage of problems (Y) with pass@1 lower than the indicated value (X).

For a given model, we define a *reliable failure* as a prompt that is in *First/Last Failure* but has pass@1 greater than 0.8 (problems to the right of the dashed line at 0.8 in the CDF). These are unlucky cases: the prompt failed when the student tried it, but turns out to be reliable. We find that GPT-3.5-Turbo-0301 and StarCoderBase have one and two reliable failures each. Similarly, we define an *unreliable success* as a prompt that is in *First/Last Success* but has pass@1 lower than 0.2. These are lucky cases: the prompt worked once for a student, but that success is hard to reproduce. We find that nearly 10% of successful prompts are unreliable for small models, but less than 3% are unreliable with larger models .

This has implications for model selection. It is not adequate to optimize a model to achieve high pass@1 on any benchmark (including StudentEval). Instead, an ideal Code LLM would both maximize pass@1 and minimize its variability.

4.3 Participant Success Rates

Examining prompt success rates by participant reveals a wide spectrum of prompting ability levels among our participants (Figure 4). Although some non-experts achieve prompt success rates over 50% with StarCoderBase, a large number struggle to write reliably successful prompts.

5 What Makes a Successful Prompt?

A participant might have a low success rate for various reasons. They might not be very skilled at writing prompts, describing the problem to be solved vaguely or even incorrectly. Or they may be writing clear explanations of the problems, but in a style that the model does not understand. Thus, a low success rate does not necessarily indicate a lack of skill on the part of the participant; it can also indicate that models systematically struggle with particular ways to describe code. In this section, we explore the various factors that impact the success of non-expert-written prompts.

5.1 Trends in Student Word Choice

To explore the relative importance of different words, we tokenized the StudentEval prompts and computed TF-IDF values for the four data subsets and then calculated the mean score per word across all prompts. Figure 5 shows the mean frequency matrix for words that appear in the top 25 for *all* subsets . The top words are a mix of English and Python terms, including many related to types, sequencing, or choice (Figure 5). The inclusion of "return" may be related to the fact that Codex seems to default to printing output, causing tests to fail; students may learn to specify "return" through experience. We see a similar trend for parameter names.

5.2 Statistical Significance of Prompt Wording

We fitted mixed-effects regression models to the data to test the impact of prompt length and wording choices. All models include random effects for problems and use StarCoderBase pass@1 rates as the response variable. For vocabulary-level features, we use indicator variables: 1 if the prompt uses the word and 0 otherwise.

Length Contrary to our expectations, we observed a statistically significant positive effect of prompt length on pass@1 rates

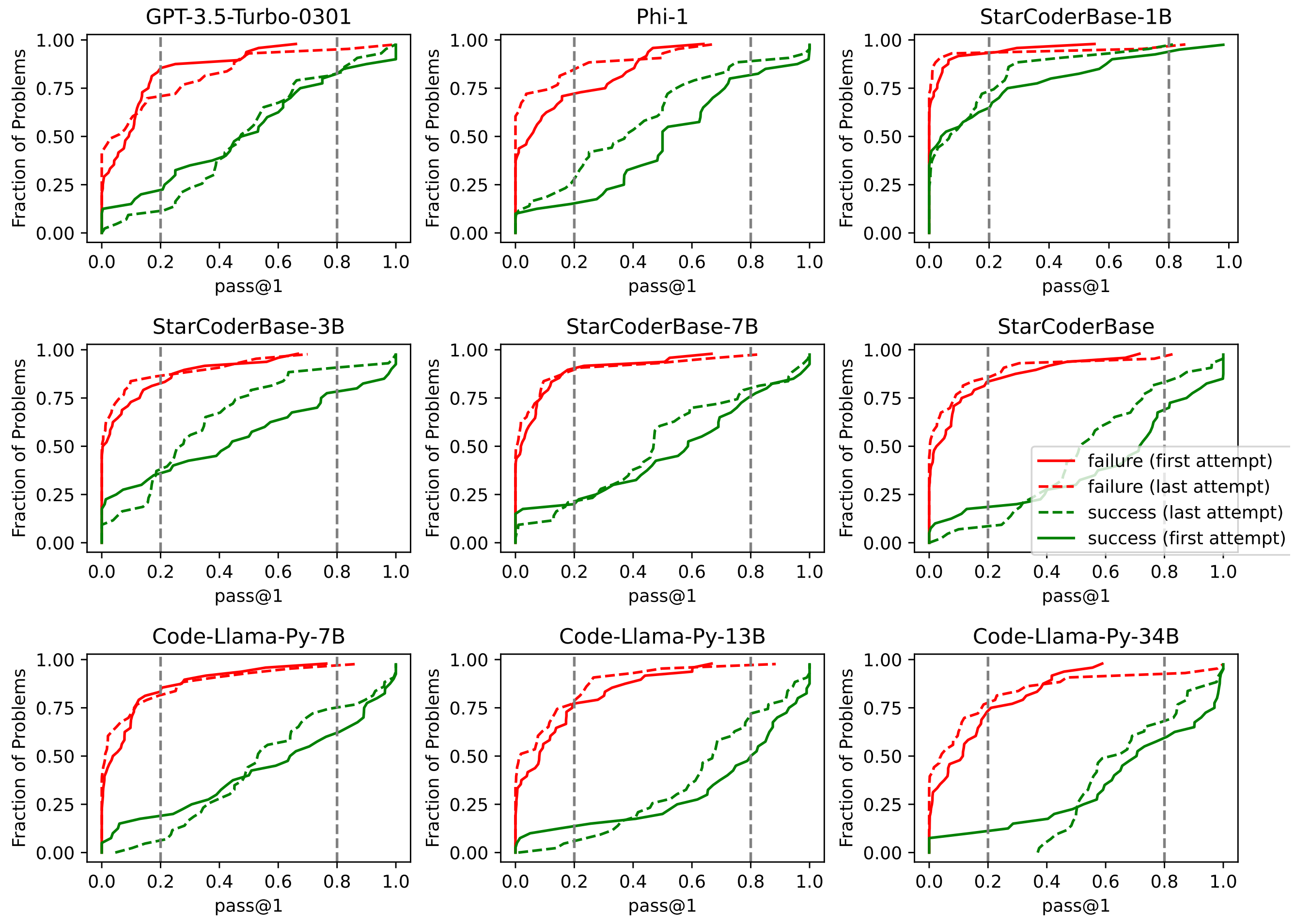


Figure 3: CDFs of mean per-problem pass@1 for Code LLMs on the four subsets of StudentEval. The y-axis shows the fraction of problems in each subset. The x-axis shows by-problem mean pass@1 for student prompts.

(p=0.007). However, this finding seems driven by last submissions, where successful prompts are on average longer; the average length is similar for passing and failing first prompts (Figure 2a). Qualitatively, we have observed that students tend to add more detail on subsequent attempts rather than modifying their earlier text, which likely contributes to this finding.

Input/output word choice We found a significant positive effect of mentioning "return" in the prompt (p<0.0001). This likely resolves the problematic ambiguity associated with prompts that mention "output" rather than specifying whether the function should return or print (Figure 6b).

Datatype mentions We explored the effect of mentioning dictionaries, lists, and number types, as well as including instances of lists and dictionaries in the prompt. We found a reliable positive effect of mentioning "list" (p=0.02), and a borderline negative effect of mentioning "array" (p=0.053). This suggests that StarCoderBase is sensitive to Python terminology conventions.

Function and parameter names We found no reliable effect of mentioning the parameter names in the prompt, but a significant negative effect of mentioning the function name (p=0.02).

5.3 Inspecting Visual Representations

We generated embeddings of each prompt from the last-layer attention weights of StarCoderBase in order to explore prompt similarities and differences. Figure 6 shows key clusters of embeddings plotted using t-SNE [26].

Multiple prompt formulations exist. Some problems form multiple clusters, indicating different ways of describing the task. The `combine` problem prompts form two clusters (Figure 6c). The top right cluster contains succinct prompts, as exemplified by Prompt 2: *Combine lists from 11 to lists from 12.* The bottom left prompts provide detailed step-by-step directions: *Takes an input of two lists, l1 and l2, each of which also contains lists. It combines the first list in l1 with the first one in l2, then continues for all items in l1 and l2. It outputs this final list which is a combination of l1 and l2* (Prompt 1).

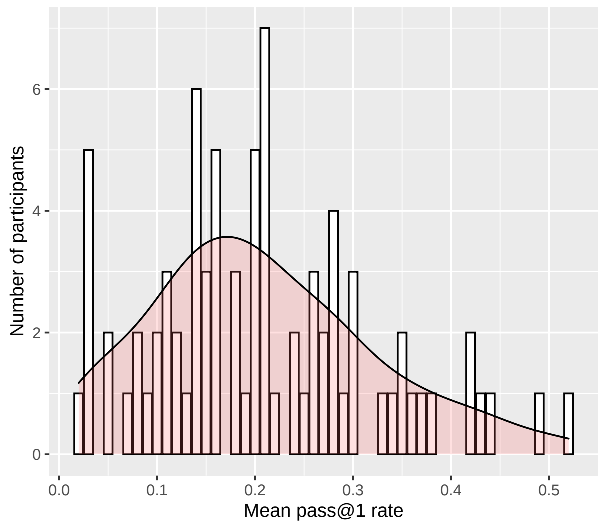


Figure 4: Participant mean pass@1 rates with StarCoderBase

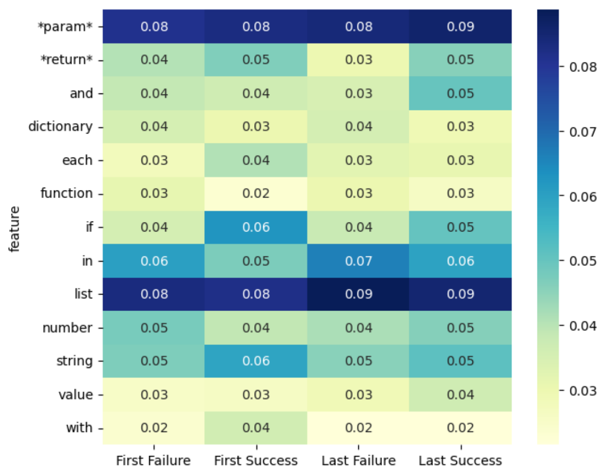


Figure 5: TF-IDF values for overlapping words in the top 25 words for all subsets.

Both approaches can generate passing programs; future work could explore whether there are style differences between the programs generated by different prompting methods.

Errors and ambiguities pattern together Examining problem sub-clusters also reveals patterns in prompting failures. In Figure 6b, for instance, there is a sub-cluster of prompts that are ambiguous about whether the function should print or return the desired value. Although a human might be able to disambiguate, these are unreliable prompts: the model may sometimes generate a solution using `print` and sometimes using `return`. Another sub-cluster consists of prompts that contain the string "aspen" (lower-case) rather than "Aspen" (upper-case), causing the generated code to fail test cases.

Certain prompting styles are challenging Although most prompt embeddings cluster by problem, a handful of clusters contain prompts for multiple problems, representing cases where the model struggles to distinguish among problem descriptions. One prompting style that students use is to describe the function's behavior in terms of expected input/output pairs. For instance, *If the number is below 10, make it 10 [...]* is a prompt that uses this strategy for `increaseScore`.

Although there are passing examples of this style, it does not seem to work for problems with more complex data, such as nested lists or dictionaries. Figure 6a shows a cluster of prompts that give examples of lists. These prompts describe different problems, yet their embeddings cluster together away from the clusters of their respective problems, indicating that the model may struggle to differentiate these rarer values. This style of prompt is likely to be well-understood by humans, yet works poorly for current Code LLMs.

5.4 Ambiguity in Prompts

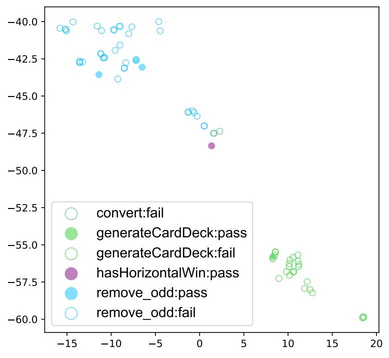
Most work on Code LLMs asks whether models produce correct code for a given prompt. However, it is also possible to have an ambiguous prompt that generates *semantically* different functions. Testing semantic equivalence is undecidable, but we can compute a lower bound on the number of semantically different functions: for each prompt completion, we use the inputs from the test cases as a vector of examples. We run each completion to collect a vector of outputs that forms the function's *test signature* [24]. When two functions have distinct test signatures, they are semantically distinct. Identical results are inconclusive.

Figure 7a summarizes results for each subset of StudentEval. As expected, Success prompts the most reliable: they generate fewer functions on average than First/Last Failure prompts. Overall, however, prompts generate a surprising number of distinct functions. Even prompts that are relatively clear to humans can be *unreliable* and generate many distinct functions (as in the example in Figure 7b).

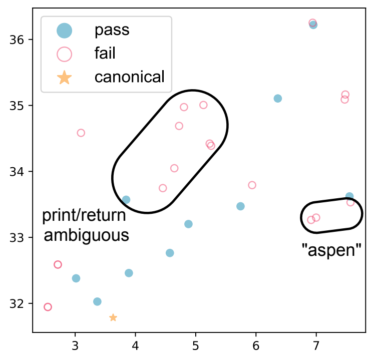
This highlights the importance of evaluating prompt reliability. Though the Figure 7b prompt produced a passing function during the experiment, it was likely to fail. This has important implications for Code LLMs as teaching tools (see Finnie-Ansley et al. [10], Leinonen et al. [17]): reliability issues may mislead students into thinking their descriptions are better than they are or into overcomplicating descriptions that are straightforward to a human, but unreliable for models.

6 Conclusion

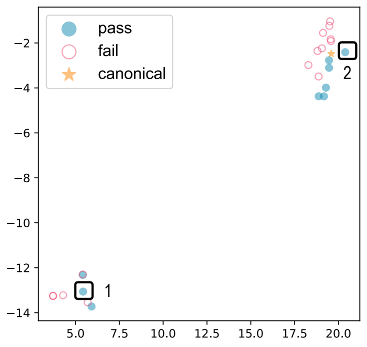
We present StudentEval, a large benchmark for Code LLMs, where the prompts are written by students who have completed one semester of Python. A key feature of StudentEval is that it has numerous different natural language descriptions of each problem, written by a key group of non-expert programmers: beginning students. We show that larger models are more capable of following student-written instructions than smaller models. We also find that many student-written prompts are unreliable (have low pass@1): students get lucky (or unlucky) when using Code LLMs. Finally, we investigate the question of what makes a good prompt from



(a) Test case cluster



(b) `check_for_aspen` prompts



(c) `combine` prompts

Figure 6: Prompt embeddings generated using StarCoderBase and reduced using t-SNE.

Subset	#Functions
failure (first attempt)	2.2 (2.0) ± 1.6
failure (last attempt)	2.4 (2.0) ± 1.6
success (first attempt)	1.9 (1.0) ± 1.3
success (last attempt)	2.2 (2.0) ± 1.3

(a) Mean (median) & standard deviation of the number of functions produced by StarCoderBase for each prompt.

The function takes a string of text as an input. For words in the string with an odd number of letters, every other letter is capitalized starting with the first letter. For words in the string with an even number of letters, every other letter is capitalized starting with the second letter.

(b) A *First Success* prompt that produces 7 functions.

Figure 7: StudentEval prompts can be ambiguous to LLMs and produce several distinct functions.

several angles, finding that models struggle to understand some valid strategies, such as giving examples of complex data.

We hope that StudentEval will make it easier to evaluate how well Code LLMs work when given descriptions written by non-experts, leading to the development of models that work better for this key group of potential users.

Limitations

Although our findings shed light on how well Code LLMs work with descriptions written by one key group of non-experts, there is more work to be done. We study only one group of non-experts (beginning students); moreover, our participants were recruited from three selective institutions within the US. Other groups of students or other populations of non-experts may use different strategies to describe code. This highlights the need for more work exploring how diverse populations of non-experts might interact with Code LLMs.

Our participants also wrote their prompts interactively while using a Codex model. It is possible that they would have revised their problems differently with a different model; this is one reason we do not emphasize comparisons between Codex and other Code LLMs in our evaluation section.

Ethics Statement

There are two main ethical concerns for this work: (1) ethical concerns about the involvement of student research participants and (2) concerns about how the dataset could be used in future work.

Our work was conducted in accordance with approval from the Brandeis University Human Research Protection Program. Potential harms to student participants were a first-class consideration in the design. We sought to address power dynamics and protect participant autonomy with a number of measures. We collected data in an opt-in manner, outside of the classroom, and with informed consent. The researcher conducting the study was not affiliated with the student participant's institution. Students were asked to complete programming assignments with familiar content and were alerted to potential discomfort caused by interacting with an AI-based tool.

All identifying information has been removed from the dataset. We plan to release the full dataset via the Open Science Framework; participants consented to the release of their anonymized data. The main data file is included as part of the Appendix. The Appendix also contains a "Datasheet for Dataset" outlining pertinent dataset information. We provide pertinent screenshots and text illustrating the experimental platform in the Appendix. The full experimental protocol will be shared publicly through the Open Science Framework upon acceptance.

Our second ethical concern is that releasing this dataset may lead to the development of technology that we would not build ourselves, such as attempts to automate education in a way that would negatively impact the educational experience of students. We feel that the benefit of providing this data, which we hope will lead to Code LLMs that work better for non-expert users, democratizing access to programming, outweigh this risk.

It is also possible that future users may generalize results from the dataset beyond what is appropriate; our study involves early CS students in a particular educational context (selective US institutions) and may not generalize to other populations.

Finally, this research was only possible due to model access and funding. To obtain the benchmark results from the 12 LLMs, we used around 2 weeks of GPU time on an H100 GPU. There are ongoing ethical concerns about access to models and infrastructure. The evaluation of the dataset in this paper centers both open-source and small-scale models, but fully addressing these issues should be a priority for the broader community.

Acknowledgements

This work is partially supported by the National Science Foundation (SES-2326173, SES-2326174, and SES-2326175).

References

[1] Aida Amini, Saadia Gabriel, Shanchuan Lin, Rik Koncel-Kedziorski, Yejin Choi, and Hannaneh Hajishirzi. 2019. MathQA: Towards Interpretable Math Word Problem Solving with Operation-Based Formalisms. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*.

[2] Ben Athiwaratkun, Sanjay Krishna Gouda, Zijian Wang, Xiaopeng Li, Yuchen Tian, Ming Tan, Wasi Uddin Ahmad, Shiqi Wang, Qing Sun, Mingyue Shang, Sujan Kumar Gonugondla, Hantian Ding, Varun Kumar, Nathan Fulton, Arash Farahani, Siddhartha Jain, Robert Giaquinto, Haifeng Qian, Murali Krishna Ramanathan, Ramesh Nallapati, Baishakhi Ray, Parminder Bhatia, Sudipta Sengupta, Dan Roth, and Bing Xiang. 2023. Multi-Lingual Evaluation of Code Generation Models. In *International Conference on Learning Representations*.

[3] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. 2021. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732* (2021).

[4] Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Ellen Tan, Yossef (Yossi) Adi, Jingyu Liu, Tal Remez, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Defossez, Jade Copet, Faisal Azhar, Hugo Touvron, Gabriel Synnaeve, Nicolas Usunier, and Thomas Scialom. 2023. Code Llama: Open Foundation Models for Code.

[5] Shraddha Barke, Michael B. James, and Nadia Polikarpova. 2023. Grounded Copilot: How Programmers Interact with Code-Generating Models. *Proceedings of the ACM on Programming Languages (PACMPL)* 7, OOPSLA (2023).

[6] Federico Cassano, John Gouwar, Daniel Nguyen, Sydney Nguyen, Luna Phipps-Costin, Donald Pinckney, Ming-Ho Yee, Yangtian Zi, Carolyn Jane Anderson, Molly Q Feldman, et al. 2023. MultiPL-E: A Scalable and Polyglot Approach to Benchmarking Neural Code Generation. *IEEE Transactions on Software Engineering* (2023).

[7] Federico Cassano, Luisa Li, Akul Sethi, Noah Shinn, , Abby Brennan-Jones, Anton Lozhkov, Carolyn Anderson, and Arjun Guha. 2024. Can It Edit? Evaluating the Ability of Large Language Models to Follow Code Editing Instructions. In *LLM4Code Workshop*.

[8] 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. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021).

[9] Zhiyu Fan, Xiang Gao, Martin Mirchev, Abhik Roychoudhury, and Shin Hwei Tan. 2023. Automated Repair of Programs from Large Language Models. In *IEEE/ACM International Conference on Software Engineering (ICSE)*.

[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* (Virtual Event, Australia) *(ACE '22)*. Association for Computing Machinery, New York, NY, USA, 10–19. https://doi.org/10.1145/3511861.3511863

[11] Suriya Gunasekar, Yi Zhang, Jyoti Aneja, Caio César Teodoro Mendes, Allie Del Giorno, Sivakanth Gopi, Mojan Javaheripi, Piero Kauffmann, Gustavo de Rosa, Olli Saarikivi, Adil Salim, Shital Shah, Harkirat Singh Behl, Xin Wang, Sébastien Bubeck, Ronen Eldan, Adam Tauman Kalai, Yin Tat Lee, and Yuanzhi Li. 2023. Textbooks Are All You Need. https://doi.org/10.48550/arXiv.2306.11644 arXiv:2306.11644 [cs]

[12] Konrad Hałas. 2013. *Cost Reduction of Mutation Testing Process in the MutPy Tool.* Ph. D. Dissertation. Instytut Informatyki.

[13] Dan Hendrycks, Collin Burns, Saurav Kadavath, Akul Arora, Steven Basart, Eric Tang, Dawn Song, and Jacob Steinhardt. 2021. Measuring Mathematical Problem Solving With the MATH Dataset. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 2)*.

[14] Yue Jia and Mark Harman. 2010. An analysis and survey of the development of mutation testing. *IEEE transactions on software engineering* 37, 5 (2010), 649–678.

[15] Sumith Kulal, Panupong Pasupat, Kartik Chandra, Mina Lee, Oded Padon, Alex Aiken, and Percy S Liang. 2019. Spoc: Search-based pseudocode to code. *Advances in Neural Information Processing Systems* 32 (2019).

[16] Yuhang Lai, Chengxi Li, Yiming Wang, Tianyi Zhang, Ruiqi Zhong, Luke Zettlemoyer, Scott Wen-tau Yih, Daniel Fried, Sida Wang, and Tao Yu. 2023. DS-1000: A Natural and Reliable Benchmark for Data Science Code Generation. In *International Conference on Machine Learning (ICML)*.

[17] Juho Leinonen, Paul Denny, Stephen MacNeil, Sami Sarsa, Seth Bernstein, Joanne Kim, Andrew Tran, and Arto Hellas. 2023. Comparing Code Explanations Created by Students and Large Language Models. In *Conference on Innovation and Technology in Computer Science Education (ITiCSE)*. Association for Computing Machinery, New York, NY, USA, 124–130. https://doi.org/10.1145/3587102.3588785

[18] Raymond Li, Loubna Ben Allal, et al. 2023. StarCoder: May the Source Be with You! *Transactions of Machine Learning Research* (Dec. 2023).

[19] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2023. Is Your Code Generated by ChatGPT Really Correct? Rigorous Evaluation of Large Language Models for Code Generation. In *Conference on Neural Information Processing Systems (NeurIPS)*.

[20] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, MING GONG, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie LIU. 2021. CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation. In *Conference on Neural Information Processing Systems Datasets and Benchmarks Track*.

[21] Niklas Muennighoff, Qian Liu, Armel Zebaze, Qinkai Zheng, Binyuan Hui, Terry Yue Zhuo, Swayam Singh, Xiangru Tang, Leandro Von Werra, and Shayne Longpre. 2023. OctoPack: Instruction Tuning Code Large Language Models. In *NeurIPS Workshop on Instruction Tuning and Instruction Following*.

[22] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2023. CodeGen: An Open Large Language Model for Code with Multi-Turn Program Synthesis. In *International Conference on Learning Representations (ICLR)*.

[23] Max Schafer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. 2024. An Empirical Evaluation of Using Large Language Models for Automated Unit Test Generation. *IEEE Transactions on Software Engineering* 50, 01 (jan 2024), 85–105. https://doi.org/10.1109/TSE.2023.3334955

[24] Abhishek Udupa, Arun Raghavan, Jyotirmoy V. Deshmukh, Sela Mador-Haim, Milo M.K. Martin, and Rajeev Alur. 2013. TRANSIT: Specifying Protocols with Concolic Snippets. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

[25] 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 *Extended Abstracts of the 2022 CHI Conference on Human Factors in Computing Systems* (New Orleans, LA, USA) *(CHI EA '22)*. Association for Computing Machinery, New York, NY, USA, Article 332, 7 pages. https://doi.org/10.1145/3491101.3519665

[26] Laurens Van der Maaten and Geoffrey Hinton. 2008. Visualizing data using t-SNE. *Journal of machine learning research* 9, 11 (2008).

[27] Albert Ziegler, Eirini Kalliamvakou, X Alice Li, Andrew Rice, Devon Rifkin, Shawn Simister, Ganesh Sittampalam, and Edward Aftandilian. 2022. Productivity assessment of neural code completion. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*. 21–29.