

CodeBenchGen: Creating Scalable Execution-based Code Generation Benchmarks

Yiqing Xie¹ Alex Xie¹ Divyanshu Sheth¹ Pengfei Liu² Daniel Fried¹ Carolyn Rosé¹

¹Carnegie Mellon University ²Shanghai Jiao Tong University

Abstract

To facilitate evaluation of code generation systems across diverse scenarios, we present CodeBenchGen, a framework to create scalable execution-based benchmarks that only requires light guidance from humans. Specifically, we leverage a large language model (LLM) to convert an arbitrary piece of code into an evaluation example, including test cases for execution-based evaluation. We illustrate the usefulness of our framework by creating a dataset, Exec-CSN, which includes 1,931 examples involving 293 libraries revised from code in 367 GitHub repositories taken from the CodeSearchNet dataset. To demonstrate the complexity and solvability of examples in Exec-CSN, we present a human study demonstrating that 81.3% of the examples can be solved by humans and 61% are rated as “requires effort to solve”. We conduct code generation experiments on open-source and proprietary models and analyze the performance of both humans and models. ¹

1 Introduction

Code generation systems assist programmers by generating code based on their instructions (Simon, 1963; Feng et al., 2020; Chen et al., 2021). To evaluate the requisite capabilities of such systems, there is a need for benchmarks that simulate a variety of scenarios, including solving algorithmic problems (Austin et al., 2021; Hendrycks et al., 2021), developing data science applications (Lai et al., 2023), implementing web applications (Oda et al., 2015), and assisting in software engineering practices (Jimenez et al., 2024). A desirable practice for evaluation of code generation systems across these and other scenarios is to execute the system-generated code and check whether it passes test cases (Chen et al., 2021; Hendrycks et al., 2021; Lai et al., 2023). Such execution-based metrics have been shown to be a more reliable indicator of functional correctness and human preference than execution-free metrics (Chen et al., 2021; Wang et al., 2023; Dibia et al., 2023).

To construct execution-based benchmarks, one approach is to extract programming problems with test cases from online platforms (Hendrycks et al., 2021; Guo et al., 2024). Such benchmarks have mainly been limited to algorithms and data structures. Another approach is to manually curate evaluation examples either created from scratch (Chen et al., 2021; Austin et al., 2021; Du et al., 2023) or based on existing code fragments (Lai et al., 2023; Wang et al., 2023; Yin et al., 2023; Zhang et al., 2024), which requires heavy human effort and thus has limited scalability. Recent work leverages GitHub repositories containing test cases as resources for the construction of benchmarks (Zhang et al., 2023; Jimenez et al., 2024). However, such methods have still required human annotators to write the instructions or set up the environment. Restricting the selection of source repositories to those with adequate test cases means that such benchmarks only represent well-developed projects contributed by professional software developers, without reflecting coding styles and practices broadly.

In this paper, we present CodeBenchGen, a framework to create custom and scalable execution-based benchmarks, only requiring light human guidance to ensure quality. As shown in Figure 1, CodeBenchGen converts each selected code fragment into an evaluation example with accompanying natural-language instructions, a list of dependencies required

¹Code available at <https://github.com/Veroniciu/CodeBenchGen>.

to set up an execution environment, the context where the generated code will be inserted, the target output, and the test cases to be used for checking the functional correctness. Specifically, we use an LLM to sandbox the source code fragment and generate test cases so that the code can be tested in an isolated environment. To ensure the quality of the generated examples, we iteratively execute and refine examples to ensure they pass all test cases while satisfying the user’s requirements.

As a demonstration of CodeBenchGen’s capabilities, we construct a benchmark, Exec-CSN, which contains 1,931 examples revised from code in 367 GitHub repositories taken from the CodeSearch-Net dataset (Husain et al., 2020). We examine the quality of the generated examples in terms of whether they are diverse, with varying degrees of complexity, but still able to be solved by programmers. Analyses show that Exec-CSN has high domain diversity, covering 293 libraries and 668 repository topics. Exec-CSN also contains examples created by diverse programmers, including both small-scale projects created by beginners and large-scale projects with over 400 professional software engineers as contributors.

We conduct a human study to verify the complexity and solvability of the generated examples. We invited computer science graduate students to complete the code in each evaluation example and then answer several survey questions. Results show that 81.3% of the examples are solvable by the participants. The problems have a range of difficulties: according to our survey questions, 39% of the examples can be “quickly solved by most programmers,” 44% “requires some effort to solve” and 17% “can only be solved by professionals or with great effort.” Participants noted that solving 50% of the examples required them to access external resources (e.g., search engines). Results also demonstrate the quality of test cases, with 85% of the test cases rated as “reasonable and not too trivial.”

A high-quality benchmark useful for driving research forward differentiates between models with different levels of capability. We conduct code generation experiments on 12 models, including proprietary models and open-source models with different sizes (7B, 34B, and 70B parameters). Results show that the best model achieves a Pass@1 score of only 37.21%, which again reveals the complexity of our dataset. Analyses show that the models receive overall lower results on examples with longer target lengths, more function calls or containing external libraries, which suggests potential directions to build more complex benchmarks. We also benchmark in a setting where both humans and models can iteratively improve their answers based on the execution results. We observe that GPT-4 has a better initial Pass@1 score, but humans benefit more from iterative improvement.

Contributions. (1) We present CodeBenchGen, a framework to convert code fragments to execution-based evaluation examples, which enables researchers to construct custom and scalable benchmarks. (2) We use our framework to create a new benchmark, Exec-CSN, from naturally occurring GitHub code and conduct analyses and human studies to demonstrate the diversity, complexity, and solvability of the generated examples. (3) We benchmark 10 code generation models on Exec-CSN and conduct comparisons and analyses on model and human performance.

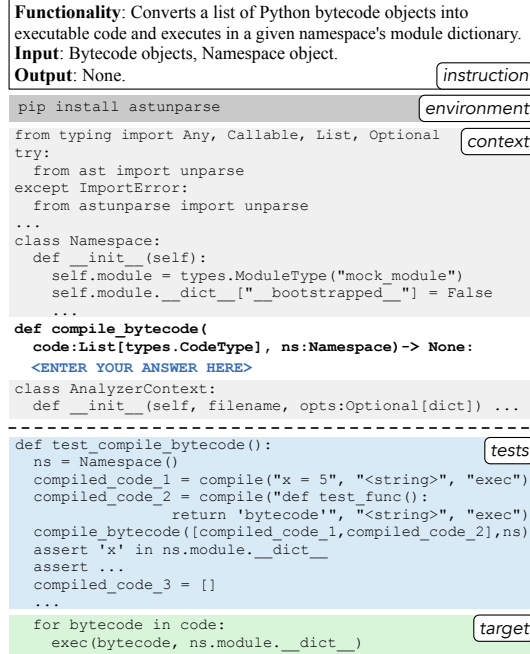


Figure 1: An evaluation example generated by CodeBenchGen. The context and target are revised from an arbitrary piece of code by an LLM. The instruction and test cases are generated from scratch based on the revised code.

2 Related Work

2.1 Code Generation Benchmarks

Researchers have built code generation benchmarks for diverse scenarios, such as query parsing (Zelle & Mooney, 1996), web development (Oda et al., 2015), problem-solving for programmers (Yin et al., 2018), etc. To improve the reliability of evaluation, HumanEval (Chen et al., 2021) evaluates code generation by executing test cases. It is followed by other execution-based benchmarks with larger scale (Hendrycks et al., 2021; Austin et al., 2021), better test coverage (Liu et al., 2023), or multiple languages (Cassano et al., 2022).

To extend execution-based evaluation to diverse scenarios, researchers have manually created benchmarks for data science code (Lai et al., 2023), Jupyter notebooks (Yin et al., 2023), user-defined classes (Du et al., 2023), etc. Recent works further enrich the code generation context by basing benchmarks on repositories with test cases (Zhang et al., 2023; Jimenez et al., 2024). However, such benchmarks require heavy human effort to set up the environment, write instructions, or even write the evaluation examples from scratch. Requiring the repositories to have existing test cases further limits the scalability and diversity of the datasets. In comparison, our method can be applied to arbitrary code fragments and only requires high-level human guidance.

2.2 Automatic Test Generation

Traditional test generation methods include black-box methods that generate random inputs to the code to test (Necula, 2000; Yang et al., 2011; Cha et al., 2015) and white-box methods that analyze the structure of the code (King, 1976; Cadar et al., 2008). Such methods lack context understanding and the generated tests are typically executable but lack relevance. For instance, it may generate a random string as input, while a JSON string is expected. Another category of methods trains language models to generate test cases (Alagarsamy et al., 2023; Tufano et al., 2021), which suffer from unsatisfactory accuracy. Similar to our test generation strategy, recent works prompt LLMs to generate, debug, and improve test cases, which empirically generates more correct unit tests than existing methods (Liu et al., 2022; Xie et al., 2023; Liu et al., 2023). Our work leverages LLM-based test generation, one of the best-performing approaches, as one of the steps of benchmark construction.

3 Methodology of CodeBenchGen

The framework of CodeBenchGen is illustrated in Figure 2. The input is a code fragment selected by the user of CodeBenchGen, and the result is an evaluation example for code generation, such as illustrated earlier in Figure 1.

Step 1: Sandboxing. To construct a user-friendly benchmark, we aim to test the generated code in an isolated environment such as a sandbox. However, it is nontrivial to directly execute an arbitrary piece of code from GitHub, which may contain complicated dependencies on external files, external API calls, access to the file system, etc.

As a result, after selecting a segment of the code as the target code, we use an LLM to sandbox the source code so that it can be executed in an isolated environment. To create examples close to real-world code, we prompt the model to remove as little source code as much as possible and add new code only when necessary. In our experiments, we observe that the model can generate new classes of functions not presented in the source code, replace external API calls with mock connections, create strings to simulate files in the system, etc. Examples can be found in Appendix A.5.

It is possible that the model does not fully follow the instructions and omits too much context or outputs an incomplete piece of code. We thus allow the users to set custom requirements for the generated examples, and automatically re-generate examples that do not satisfy the requirements. In our experiments, we set requirements on the length of the generated context and the target code.

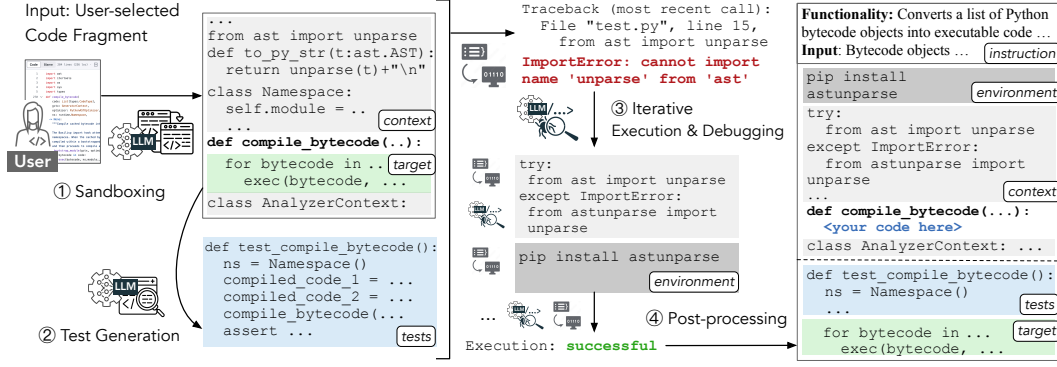


Figure 2: The framework of CodeBenchGen, which leverages an LLM to convert a code fragment selected by the user to an evaluation example. The framework sandboxes the code fragment to run in an isolated environment and iteratively debugs or regenerates the code to ensure it is functionally correct and follows the user’s requirements.

Step 2: Test Generation. In the second step, the input is code generated in step 1 and we call an LLM to generate test cases to verify the functionality of the generated code. To encourage test coverage, we prompt the model to generate at least three test cases. Similar to step 1, we check whether the generated tests call the target code and contain at least three assert statements, and re-generate if not.

Step 3: Iterative Execution and Debugging. Since the code generated in steps 1 and 2 could have errors, we iteratively execute the code scripts and use an LLM to debug the code until the target output is able to pass all test cases. Unlike previous work that only aims to generate and debug the tests (Xie et al., 2023), we allow the LLM to debug the entire generated code, including the target output, the context, and the tests. Similar to the first two steps, to avoid the revised code deviating too much from the input, we re-run debugging for the examples where the final generated code has incomplete tests or has a much shorter length than the original input of this step.

Step 4: Post-processing. After generating the code, we **generate natural language instructions** for the examples. As discovered by previous work (Wen et al., 2024), adding I/O specifications (i.e., the descriptions of the input and output of the target code) consistently improves model performance. We thus use the LLM to generate instructions with natural language descriptions of the desired functionality, input, and output of the target code.

Additionally, we use an LLM to **generate additional tests** (Liu et al., 2023). In evaluation, we execute each set of tests separately and check whether all of them are passed, thereby achieving strictly better test coverage than the initial tests. We note that there are other test generation methods such as fuzzing methods and learning-based methods. As discussed in §2.2, we choose LLM-based methods for its high accuracy.

In previous steps, we sequentially installed the dependencies for each evaluation example, which could potentially conflict with each other. Hence, to set up a shared runtime environment where all of the examples can run without dependency conflicts, we **conduct a final filtering pass**. To do this, we restart the sandbox, install the dependency list, and execute all examples again. The examples with compilation or runtime errors are filtered out. We further filter out examples containing potentially destructive or stateful code. More details on our filtering strategy can be found in Appendix A.1

4 Creating a Dataset: Exec-CSN using CodeBenchGen

To demonstrate the scalability of our method, we create a dataset, Exec-CSN. Examples from our Exec-CSN dataset can be found in Appendix A.5. We take the source code fragment from the CodeSearchNet (Husain et al., 2020) dataset, a function completion dataset that does not support execution-based evaluation, where the input contains the function signature and

the docstring, and the target is the function body. We choose CodeSearchNet as the source because of its large scale and coverage across a diversity of examples from GitHub.

We sampled 5,000 Python examples from the test split of CodeSearchNet. Because the function signature only contains limited information, we downloaded the corresponding code file from GitHub as additional context for each example. We filtered out examples that do not have a valid file link. To simplify the execution environment of our benchmark, we filter out examples with I/O operations by keywords. There were 4,079 examples left after filtering.

We take the 4,079 examples as the input and run our framework. In the first three steps, we re-generate each example at most three times and run the execution and debugging cycle for at most three iterations per example. We use GPT-4 as the LLM due to its high quality. In post-processing, we use GPT-3.5 to generate 5 more sets of tests for each example and filter out incorrect ones. We choose GPT-3.5 to reduce the bias towards GPT-4 in the test cases. Details and experiments on test generation can be found in Appendix A.1 and A.2. Table 1 shows the number of evaluation examples generated in each step. After each iteration of debugging, our framework creates more examples where the target can pass all test cases. We finally obtained 1,931 examples after post-processing: successfully converting 47% of the source code fragment into evaluation examples with executable test cases. Since we take the source code from an existing dataset, the only human effort in constructing Exec-CSN is to make a set of requirements for the examples and write corresponding prompts, which does not grow with the dataset size.

Step	Description	# Examples
① ②	Sandboxing & Test Gen	1,260
③	1st Exec & Debug Iter	1,973
	2nd Exec & Debug Iter	2,155
	3rd Exec & Debug Iter	2,343
④	Post-processing	1,931

Table 1: Number of evaluation examples generated in each step of our framework. We only count examples where the target can pass all the test cases. We have 4,079 pieces of code in the input.

5 Quality Verification for Exec-CSN

We evaluate the quality of the generated benchmark with both a corpus-based evaluation and a human study. In particular, we study three research questions: **(RQ1-Diversity)** Can our framework generate diverse evaluation examples? **(RQ2-Complexity)** Do the examples have varying degrees of complexity? **(RQ3-Solvability)** Are the examples possible to solve, including reasonable instructions, code, and test cases?

5.1 Diversity Analysis by Corpus-based Evaluation

To answer RQ1-Diversity, we compare the diversity in (1) the domains of code and (2) the programmers creating the code using a corpus-based analysis. Table 2 presents the statistics of Exec-CSN and existing benchmarks, which shows that Exec-CSN covers more diverse libraries (both standard and external), repository topics, and number of contributors.

Analysis of Domain Diversity. To estimate the most common domains, we leverage the Stack dataset (Kocetkov et al., 2022), which contains 317M code files from GitHub, and count

Dataset	#Examples	#Repo	#Topics	#Libs (Std.+Ext.)	#Contributors	Source
HumanEval (Chen et al., 2021)	164	–	–	4 = 4 + 0	–	Hand-written
MBPP (Austin et al., 2021)	974	–	–	12 = 12 + 0	–	Hand-written
APPS (Hendrycks et al., 2021)	10,000	–	–	0	–	Coding contest
DS1000 (Lai et al., 2023)	1,000	–	–	7 = 0 + 7	–	Stack Overflow
ODEX (Wang et al., 2023)	945	–	–	79 = 48 + 31	–	Stack Overflow & Hand-written
ClassEval (Du et al., 2023)	100	–	–	28 = 19 + 9	–	Hand-written
CoderEval (Zhang et al., 2024)	230	43	145	179 = 86 + 93	1 ~ 417	GitHub
RepoEval-func (Zhang et al., 2023)	455	8	23	75 = 33 + 42	1 ~ 51	GitHub
SWE-BENCH (Jimenez et al., 2024)	2,294	12	60	214 = 111 + 103	185 ~ 444	GitHub
Exec-CSN (ours)	1,931	367	668	293 = 118 + 175	1 ~ 449	GitHub

Table 2: Statistics of Exec-CSN compared to existing execution-based code generation datasets. RepoEval-func stands for the function completion task of RepoEval. We compare the repository number, topic number, and contributor number for datasets built from GitHub. We count the libraries in the oracle context for RepoEval-func and SWE-BENCH.

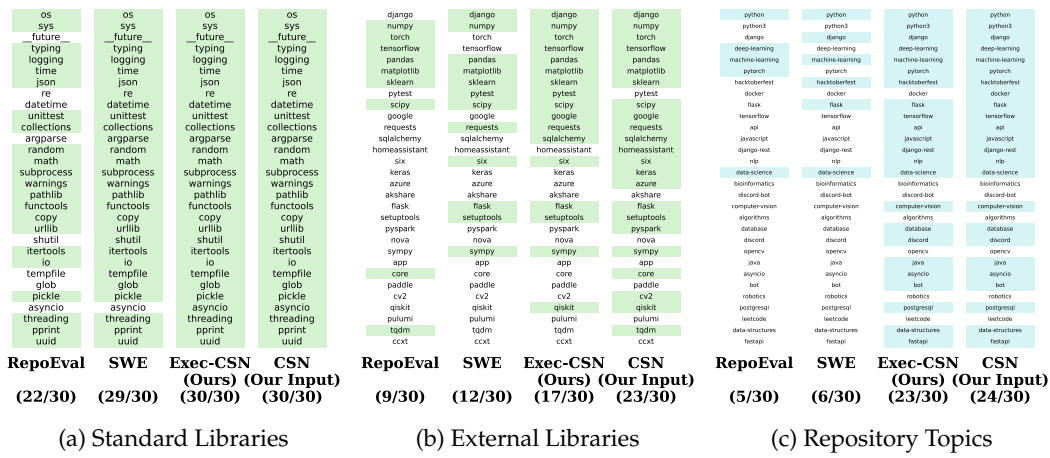


Figure 3: Domain diversity in different datasets. We choose the top-30 most common libraries or topics, estimated by frequency in the Stack dataset (Kocetkov et al., 2022), and check whether they are present in each dataset. “CSN” denotes CodeSearchNet, which we use as input to our framework to create Exec-CSN.

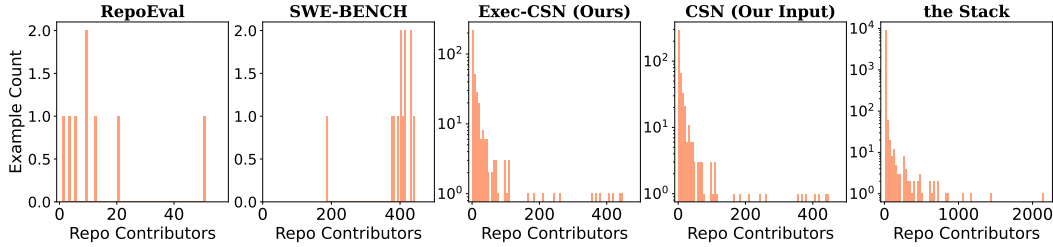


Figure 4: Distribution of contributor numbers of the repositories in different datasets.

the libraries and repository topics in 10K randomly sampled Python files. As shown in Figure 3, among the top-30 most common standard/external libraries and topics, Exec-CSN has higher coverage than existing execution-based benchmarks. For instance, RepoEval does not contain `django`, the top-1 external library, and SWE-BENCH misses `torch` and `tensorflow`, which are ranked in the 3rd and 4th places. In comparison, Exec-CSN covers all top-10 external libraries. We can also observe that Exec-CSN retains 30/30 standard libraries, 17/23 external libraries, and 23/24 repository topics from its input code, which suggests that our framework can preserve the diversity in the input to a large extent.

Analysis of Programmer Diversity. Figure 4 shows the distribution of the number of contributors to each repository across benchmarks. We observe that compared to existing benchmarks, the distribution of Exec-CSN is closer to the natural distribution evidenced in the Stack. The reason is that RepoEval and SWE-BENCH rely on repositories that already contain test cases, hence both datasets skew toward projects that have high quality, large scale, and more contributors. In comparison, Exec-CSN covers repositories created by diverse contributors, reflecting a wider range of scenarios of coding.

5.2 Complexity Analysis by Corpus-based Evaluation

To investigate RQ2-Complexity, we measure the number of code tokens, depth of AST, and number of variables in both the target and the full example. We count the tokens with the GPT-4 tokenizer and parse the AST using the `tree-sitter` library. Results in Table 3 show that our framework can generate relatively long examples, with an average of 496.53 code tokens, 12.03 variables, and 9.38 layers in the AST. The generated examples also have varying complexity, from 83 to 1,529 code tokens.

5.3 Complexity and Solvability Analysis by Human Study

We conduct a human study to study RQ2-Complexity and RQ3-Solvability.

	Context + Target			Target		
	#Code Tokens	AST Depth	# Variables	#Code Tokens	AST Depth	# Variables
Avg	491.88	9.38	12.03	73.83	7.81	4.21
[min, max]	[83, 1529]	[6, 18]	[0, 50]	[3, 714]	[3, 18]	[0, 23]

Table 3: Complexity of our dataset measured by the number of code tokens, depth of AST, and number of variables in the context and the target code.

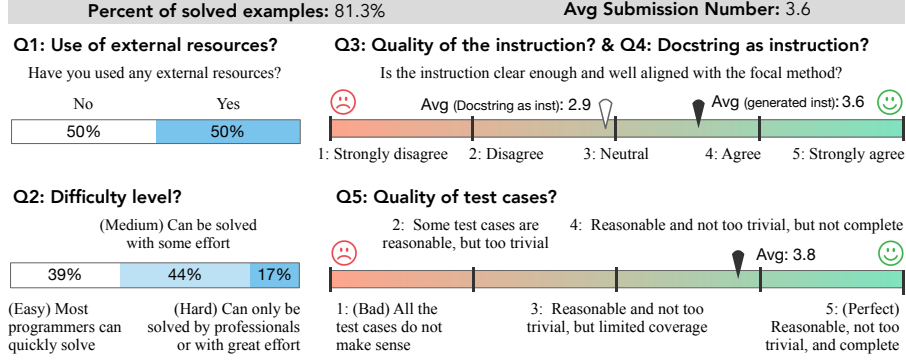


Figure 5: Human study results collected from 64 examples in Exec-CSN.

Setup. We invited computer science graduate student volunteers for the study and obtained results on 64 examples in total. To check the examples’ solvability, we present each evaluation example as a coding problem to the participants and ask them to write code based on the instruction and the context. The participants can use any external resources (e.g., search engines) except AI models. After submitting an answer, they will see the execution results of the test cases and can choose to revise their answers accordingly. We record the percentage of solved problems. Intuitively, a participant solving a problem indicates that the instruction and test cases are consistent with their target code. The use of external resources and the number of revisions indicate the complexity level of the examples.

To check the examples’ complexity, after the participants pass all test cases or decide not to proceed, we ask them to rate the level of difficulty, the clarity of instructions, and the quality of test cases using a five-point Likert scale. In past work, (Husain et al., 2020) used the docstrings of the target functions as instructions, so we also ask the participants to rate the quality of the docstrings by considering them as instructions.

Main Results. Figure 5 presents the human study results. 81.3% of the examples were solved by their assigned participant, which indicates that these examples have clear instructions, reasonable test cases, and have difficulty within a reasonable range. The average rating of our generated instructions is much higher than that of the original docstrings of the target code, which were used as instructions in previous methods (Husain et al., 2020). 85% of the test cases are rated “reasonable and not trivial.”

We also observe that the generated examples have varying complexity levels. 34.4% of the examples are solved in the first submission and 68.8% within five submissions. On 50% of the examples, the participants used external resources i.e., searching for some external libraries they are not familiar with or functionalities they want to implement. According to the self-report ratings, 39% of the examples can be quickly solved by most programmers. 44% require some effort and 17% can only be solved by professionals or with great effort.

6 Code Generation Performance Evaluation

This section aims to answer: (RQ4) How do code generation models perform on Exec-CSN?

6.1 Experimental Setup

Code Generation Models. We conduct experiments on 10 open-source and 2 proprietary models. We select open-source models across various parameter sizes (7B, 33B, 70B) and model families (Mistral, Llama, CodeQwen, and DeepSeek-Coder).

Model Family	Size	Model	Pass@1	Pass@2	Pass@5	Pass@10
Mistral	7B	OpenChat-3.5	23.61	27.08	31.39	34.34
Llama	8B	Llama-3	25.31	28.70	32.29	34.66
CodeQwen	7B	CodeQwen	28.58	32.54	36.82	39.49
DeepSeek-Coder	6.9B	DeepSeek-Coder-v1.5	28.68	32.36	36.52	39.22
DeepSeek-Coder	6.7B	Magocoder-S-DS	30.73	34.92	39.56	42.47
DeepSeek-Coder	33B	WizardCoder	33.35	36.81	40.51	42.76
DeepSeek-Coder	33B	DeepSeek-Coder	<u>34.00</u>	<u>37.69</u>	41.59	44.06
CodeLlama	34B	CodeLlama	26.24	30.11	34.37	36.95
CodeLlama	34B	Speechless-CodeLlama	32.23	36.03	40.16	42.80
CodeLlama	70B	CodeLlama	30.92	37.36	43.60	47.18
–	–	GPT-3.5	32.56	35.73	39.14	41.26
–	–	GPT-4	37.21	39.85	42.67	44.53

Table 4: Code generation results on Exec-CSN. We put models with different sizes in different groups and split open-source and proprietary models. The model with the best Pass@k performance is highlighted in **bold** and the best open-source model is underlined.

Evaluation Metrics. We measure functional correctness using the standard pass@k metric for $k \in \{1, 2, 5, 10\}$ (Chen et al., 2021).

Experimental Details. To compute Pass@k, we take 20 samples per example for open-source models and 10 samples per example for closed-source models. For all models, we sample outputs with a temperature of 0.3 and top- p of 0.95. Note that we sample with a relatively low temperature as we are primarily interested in Pass@k scores for low k ; sampling with higher temperature would likely have yielded higher Pass@{5, 10} scores across all models (Chen et al., 2021; Liu et al., 2023). In our prompts, we provide the surrounding context, function header, and docstring, but do not provide the test cases. An example prompt is shown in Appendix A.6.

6.2 Main Results

Table 4 presents the code generation results of open-source and proprietary models. The best model (GPT-4) only achieves 37.21 Pass@1, indicating that there is still room for improvement on our dataset. Comparison between open-source and proprietary models shows that WizardCoder-33B and DeepSeek-Coder consistently surpass GPT-3.5 on all metrics, although both models are still outperformed by GPT-4. Comparison between models with different sizes shows that both DeepSeek-Coder and CodeLlama greatly benefit from larger parameter sizes.

Results show that CodeLlama-70B achieves the highest Pass@5 and Pass@10 scores, surpassing GPT-4, but does not have a particularly high Pass@1. We observe that although the generated code has high quality, in around 8% of the cases, CodeLlama-70B misinterprets our queries as “dangerous” or as requests to do students’ programming homework for them and “refuses” to generate the code, which substantially drag down the Pass@1 score. This is most likely a byproduct of CodeLlama’s alignment training, which encourages it to reject dangerous or unethical requests (Rozière et al., 2024).

6.3 Results Analysis

Performance Breakdown. To identify what kind of examples are hard for models, we focus on DeepSeek-Coder-33B and GPT-4, the best open-source and proprietary model under Pass@1, and study four factors that could affect model performance: target length (number of code tokens), context length, number of function calls, and libraries. We split the examples into 5 groups based on each factor such that each group has a similar number of examples. Then we compute the models’ average Pass@1 on each group of examples.

As shown in Figure 6, when the target output has a longer length or has more function calls, both models have lower performance and the gap between DeepSeek-Coder and GPT-4 becomes larger. Additionally, both models have lower performance on examples that import any libraries, especially external libraries. However, the context length has a positive but weaker influence on model performance.

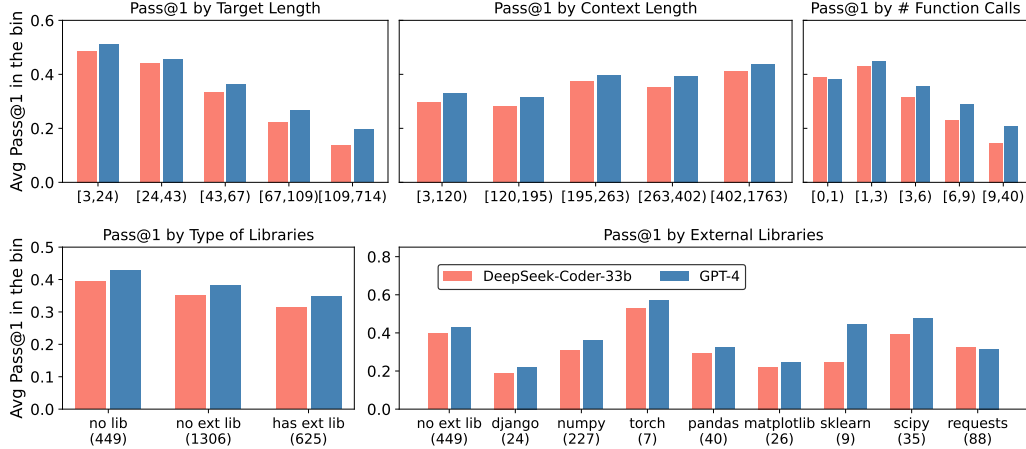


Figure 6: Code generation performance of two models on different groups of examples.

We also check the models’ performance on examples with different libraries. We inspect the 10 most frequent libraries in the stack, excluding `tensorflow` and `google` as they are present only 4 and 1 times in our dataset, respectively. Results indicate that both the difficulty of generating code and the gap between the two models varies a lot across libraries. For instance, GPT-4 surpasses DeepSeek-Coder substantially on `sklearn` but is slightly outperformed on `requests`. These results motivate benchmarks with high domain diversity to have a more comprehensive ranking of model performance.

Humans vs. Models. To compare the performance of humans and models, we implement a generation setting similar to the human study that also allows the model to revise from its previous outputs. Specifically, if the model’s output fails to pass some test cases, we append its output and the execution results of the test cases to the input and generate again.

We compare human performance to GPT-4 (OpenAI, 2023), the strongest model in our experiments (see §6.2 for details). As shown in Table 5, with no revisions, GPT-4 has a higher Pass@1 than humans. We suspect that the model is trained on code with diverse domains, while humans have limited experience in some domains and have trouble solving the problem at the first attempt. However, when revision is allowed, humans can solve much more examples than GPT-4, suggesting that humans make better use of the error messages. Our case study in Appendix A.3 further shows that GPT-4 generates more detailed and complete solutions at the first attempt, while humans write the code with an incremental approach. The revisions made by humans are also more closely related to the error message.

# Revision	GPT-4	Human
0 (Pass@1)	40.63	34.38
1	50.00	43.75
2	51.56	57.81
3	57.81	60.94
4	57.81	68.75
5-17	–	81.25

Table 5: The performance of humans and GPT-4 on the 64 examples in the human study. We show the accuracy (percentage of solved examples) after each round of revision. The accuracy with no revision is the same as Pass@1.

7 Conclusion

We presented a framework, CodeBenchGen, to assist researchers in constructing scalable and custom execution-based code generation benchmarks. The framework leverages LLMs to convert arbitrary pieces of code into evaluation examples complete with test cases. We use our framework to create a benchmark, Exec-CSN, with 1,931 examples taken from the CodeSearchNet dataset. We conduct corpus-based evaluation and a human study to verify the quality of generated examples in terms of diversity, complexity, and solvability. Compared to existing code generation benchmarks, Exec-CSN covers more diverse libraries, topics, and number of contributors. We benchmarked open-source and proprietary code generation models on Exec-CSN and analyzed both human and model performance. Results indicate that there is still room for improvement for models on our dataset.

8 Acknowledgement

We thank all participants in our human study. This work was supported in part by NSF grant DSES 2222762.

References

- Saranya Alagarsamy, Chakkrit Tantithamthavorn, and Aldeida Aleti. A3test: Assertion-augmented automated test case generation, 2023.
- Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. Program synthesis with large language models, 2021.
- Cristian Cadar, Daniel Dunbar, and Dawson Engler. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation, OSDI’08*, pp. 209–224, USA, 2008. USENIX Association.
- Federico Cassano, John Gouwar, Daniel Nguyen, Sydney Nguyen, Luna Phipps-Costin, Donald Pinckney, Ming-Ho Yee, Yangtian Zi, Carolyn Jane Anderson, Molly Q Feldman, Arjun Guha, Michael Greenberg, and Abhinav Jangda. Multipl-e: A scalable and extensible approach to benchmarking neural code generation, 2022.
- Sang Kil Cha, Maverick Woo, and David Brumley. Program-adaptive mutational fuzzing. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy, SP ’15*, pp. 725–741, USA, 2015. IEEE Computer Society. ISBN 9781467369497. doi: 10.1109/SP.2015.50. URL <https://doi.org/10.1109/SP.2015.50>.
- 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. Evaluating large language models trained on code, 2021.
- Victor Dibia, Adam Fourney, Gagan Bansal, Forough Poursabzi-Sangdeh, Han Liu, and Saleema Amershi. Aligning offline metrics and human judgments of value for code generation models. In *Findings of the Association for Computational Linguistics: ACL 2023*. Association for Computational Linguistics, 2023. URL <https://aclanthology.org/2023.findings-acl.540>.
- Xueying Du, Mingwei Liu, Kaixin Wang, Hanlin Wang, Junwei Liu, Yixuan Chen, Jiayi Feng, Chaofeng Sha, Xin Peng, and Yiling Lou. Classeval: A manually-crafted benchmark for evaluating llms on class-level code generation, 2023.
- Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. CodeBERT: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*. Association for Computational Linguistics, 2020. URL <https://aclanthology.org/2020.findings-emnlp.139>.
- Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y. Wu, Y. K. Li, Fuli Luo, Yingfei Xiong, and Wenfeng Liang. Deepseek-coder: When the large language model meets programming – the rise of code intelligence, 2024.

-
- Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and Jacob Steinhardt. Measuring coding challenge competence with apps, 2021.
- Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. Codesearchnet challenge: Evaluating the state of semantic code search, 2020.
- Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. SWE-bench: Can language models resolve real-world github issues? In *The Twelfth International Conference on Learning Representations*, 2024. URL <https://openreview.net/forum?id=VTF8yNQM66>.
- James C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, jul 1976. ISSN 0001-0782. doi: 10.1145/360248.360252. URL <https://doi.org/10.1145/360248.360252>.
- Denis Kocetkov, Raymond Li, Loubna Ben Allal, Jia Li, Chenghao Mou, Carlos Muñoz Ferrandis, Yacine Jernite, Margaret Mitchell, Sean Hughes, Thomas Wolf, Dzmitry Bahdanau, Leandro von Werra, and Harm de Vries. The stack: 3 tb of permissively licensed source code, 2022.
- Yuhang Lai, Chengxi Li, Yiming Wang, Tianyi Zhang, Ruiqi Zhong, Luke Zettlemoyer, Wentau Yih, Daniel Fried, Sida Wang, and Tao Yu. Ds-1000: a natural and reliable benchmark for data science code generation. In *Proceedings of the 40th International Conference on Machine Learning*, ICML’23. JMLR.org, 2023.
- Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation, 2023.
- Zhe Liu, Chunyang Chen, Junjie Wang, Xing Che, Yuekai Huang, Jun Hu, and Qing Wang. Fill in the blank: Context-aware automated text input generation for mobile gui testing, 2022.
- Haipeng Luo, Qingfeng Sun, Can Xu, Pu Zhao, Jianguang Lou, Chongyang Tao, Xiubo Geng, Qingwei Lin, Shifeng Chen, and Dongmei Zhang. Wizardmath: Empowering mathematical reasoning for large language models via reinforced evol-instruct, 2023.
- George C. Necula. Translation validation for an optimizing compiler. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*. Association for Computing Machinery, 2000. URL <https://doi.org/10.1145/349299.349314>.
- Yusuke Oda, Hiroyuki Fudaba, Graham Neubig, Hideaki Hata, Sakriani Sakti, Tomoki Toda, and Satoshi Nakamura. Learning to generate pseudo-code from source code using statistical machine translation. In *Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, ASE ’15, pp. 574–584, Lincoln, Nebraska, USA, November 2015. IEEE Computer Society. ISBN 978-1-5090-0025-8. doi: 10.1109/ASE.2015.36. URL <https://doi.org/10.1109/ASE.2015.36>.
- OpenAI. Gpt-4 technical report, 2023.
- Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. Code llama: Open foundation models for code, 2024.
- Herbert A. Simon. Experiments with a heuristic compiler. *J. ACM*, 1963. URL <https://doi.org/10.1145/321186.321192>.
- Michele Tufano, Dawn Drain, Alexey Svyatkovskiy, Shao Kun Deng, and Neel Sundaresan. Unit test case generation with transformers and focal context, 2021.

-
- Guan Wang, Sijie Cheng, Xianyuan Zhan, Xiangang Li, Sen Song, and Yang Liu. Open-chat: Advancing open-source language models with mixed-quality data. In *The Twelfth International Conference on Learning Representations*, 2024. URL <https://openreview.net/forum?id=A0JyfhWYHf>.
- Zhiruo Wang, Shuyan Zhou, Daniel Fried, and Graham Neubig. Execution-based evaluation for open-domain code generation. In *Findings of the Association for Computational Linguistics: EMNLP 2023*. Association for Computational Linguistics, 2023. URL <https://aclanthology.org/2023.findings-emnlp.89>.
- Yuxiang Wei, Zhe Wang, Jiawei Liu, Yifeng Ding, and Lingming Zhang. Magicoder: Source code is all you need. *arXiv preprint arXiv:2312.02120*, 2023.
- Yeming Wen, Pengcheng Yin, Kensen Shi, Henryk Michalewski, Swarat Chaudhuri, and Alex Polozov. Grounding data science code generation with input-output specifications, 2024.
- Zhuokui Xie, Yinghao Chen, Chen Zhi, Shuiguang Deng, and Jianwei Yin. Chatunitest: a chatgpt-based automated unit test generation tool, 2023.
- Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in c compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '11*, pp. 283–294, New York, NY, USA, 2011. Association for Computing Machinery. ISBN 9781450306638. doi: 10.1145/1993498.1993532. URL <https://doi.org/10.1145/1993498.1993532>.
- Pengcheng Yin, Bowen Deng, Edgar Chen, Bogdan Vasilescu, and Graham Neubig. Learning to mine aligned code and natural language pairs from stack overflow. In *International Conference on Mining Software Repositories, MSR*. ACM, 2018.
- Pengcheng Yin, Wen-Ding Li, Kefan Xiao, Abhishek Rao, Yeming Wen, Kensen Shi, Joshua Howland, Paige Bailey, Michele Catasta, Henryk Michalewski, Oleksandr Polozov, and Charles Sutton. Natural language to code generation in interactive data science notebooks. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Association for Computational Linguistics, 2023. URL <https://aclanthology.org/2023.acl-long.9>.
- John M. Zelle and Raymond J. Mooney. Learning to parse database queries using inductive logic programming. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence - Volume 2, AAAI'96*. AAAI Press, 1996.
- Fengji Zhang, Bei Chen, Yue Zhang, Jacky Keung, Jin Liu, Daoguang Zan, Yi Mao, Jian-Guang Lou, and Weizhu Chen. RepoCoder: Repository-level code completion through iterative retrieval and generation. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, 2023. URL <https://aclanthology.org/2023.emnlp-main.151>.
- Yakun Zhang, Wenjie Zhang, Dezhi Ran, Qihao Zhu, Chengfeng Dou, Dan Hao, Tao Xie, and Lu Zhang. Learning-based widget matching for migrating gui test cases. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE '24*. ACM, February 2024. doi: 10.1145/3597503.3623322. URL <http://dx.doi.org/10.1145/3597503.3623322>.
- Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric P. Xing, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. Judging llm-as-a-judge with mt-bench and chatbot arena, 2023.

A Appendix

A.1 Dataset Post-Processing Details

In Step 4 of our CodeBenchGen framework (see §3), we perform post-processing on our generated evaluation examples to validate and improve their quality. In this step, in addition to generating natural language instructions with I/O specifications, we filter the data to ensure all examples can run *safely* and generate additional test cases to more accurately measure functional correctness.

Dataset Post-Filtering. After all examples have been generated and all dependencies installed, we filter the examples based on two criteria: (1) whether the ground truth code is still able to pass generated tests and (2) whether the example contains potentially dangerous code (e.g., code that kills an OS process).

For the latter, we use simple keyword-based filtering to ban destructive operations, such as killing processes or deleting files. The full list of keywords is shown in Table 6. We find this to be sufficient for our Exec-CSN experiments. However, the keyword list is not complete. An operation not on this list or an external API that invokes one of these operations may still trigger undesirable side effects. Moreover, creating this list of keywords may require some level of domain expertise. We leave the task of creating more secure execution environments to future work.

os.kill	terminate	subprocess.call(['kill',
subprocess.call(['rm',	subprocess.call(['rmdir',	subprocess.call(["kill",
subprocess.call(["rm",	subprocess.call(["rmdir",	sys.exit
os.unlink	.unlink	.rmdir
os.remove	os.removedirs	os.rmdir
os.system	rmtree	send2trash
open(	.read	.write
.load	.dump	shutil.
glob.	os.path.	os.remove(
os.rename(	os.rmdir(	os.mkdir(
os.makedirs(	os.listdir(	.readlines(
.writelines(	.seek(	.tell(

Table 6: List of banned keywords for post-filtering

Test Augmentation Details. Execution-based evaluation is heavily dependent on the quality of the executed tests. To improve the reliability of our evaluation, we generate additional tests in post-processing. As explained in §2.2, fuzzing-based methods and learning-based methods for test generation suffer from unsatisfactory relevance or accuracy. As such, we adopt an LLM-based approach similar to that of Liu et al. (2023), where we feed the example being tested to an LLM and prompt it to generate an additional set of tests. To ensure the correctness of generated tests, we discard the sets of tests that the target code cannot successfully pass. To achieve better test coverage, we do not replace the original tests with the generated ones. Instead, we execute the tests separately and the system under test must be able to pass both.

For simplicity, instead of performing iterative execution and self-debugging as in Step 3 of our CodeBenchGen framework, we directly sample k tests from the LLM, and select the first of these k tests that is consistent with the target (i.e., the target code passes the tests). If no test is consistent with an example’s target code, we do not augment that particular example’s tests. In our experiments, we use GPT-3.5 as the test generator as it is relatively inexpensive to sample from while still yielding high-quality outputs. We sample $k = 5$ completions with a temperature of 0.3 and top- p of 0.7.

A.2 Experiments on Test Augmentation

Test Coverage Analyses. To study the quality of tests augmented by GPT-3.5 in post-processing, we compute the line coverage rate and the performance of models under the original and augmented tests. As shown in Table 7, while the original tests already display high coverage of the target code, they can be further improved by the addition of GPT-3.5-generated tests. Moreover, we find that the augmented tests are able to catch more errors as the Pass@1 scores of our top models drop significantly. Interestingly, however, after adding the new tests, the Pass@1 scores of different models all decrease by roughly the same amount and hence the overall ranking of models remains the same.

To study the test generation ability of different models, we additionally augment the tests with two other models: DeepSeek-Coder-33B and DeepSeek-Coder-7B. As shown in Table 7, combining the tests generated by either model improves the test coverage rate. Among the models, GPT-3.5 generates correct tests for the largest percentage of examples with the highest line coverage rate. Additionally, the coverage of tests generated by DeepSeek-Coder-33B is substantially higher than that of DeepSeek-Coder-7B, suggesting that the test generation ability of the models may also benefit from larger model sizes.

Note that we only keep the tests augmented by GPT-3.5 in Exec-CSN (in addition to the tests generated by GPT-4), as adding tests augmented by DS-33B and DS-7B only leads to marginal improvement in line coverage rate.

Test Cases	% Aug	% Line Coverage	Pass@1			
			GPT-4	DS-33B	GPT-3.5	DS-7B
Original (GPT-4)	-	94.18	42.53	38.27	37.05	33.07
Orig. + GPT-3.5	70.42	95.76	37.21	34.00	32.56	28.68
Orig. + DS-33B	73.33	95.32	39.79	35.77	34.66	30.54
Orig. + DS-7B	52.26	95.04	41.46	37.82	36.81	32.51
All tests	88.95	96.44	33.46	30.56	30.01	25.57

Table 7: Line coverage of the target code and downstream Pass@1 performance of the models under the original and augmented tests. The original tests are generated by GPT-4. “% Aug” is the percentage of examples for which we successfully generate additional tests.

Analyses of Self-bias. Previous work shows that a wide range of LLMs have self-bias, which means the models rank their own generation output higher than other models’ output (Zheng et al., 2023). To study whether the tests generated by LLMs also have self-bias, we check the code generation performance of GPT-3.5 and GPT-4 on the tests generated by GPT-3.5 and GPT-4, separately. We only experiment on the examples where we successfully generate tests by GPT-3.5.

As shown in Table 8, the Pass@1 of GPT-4 is 6.59% higher than GPT-3.5 on GPT-4 generated tests, but is only 3.01% higher on GPT-3.5 generated tests. This suggests that the tests generated by LLMs may also have self-bias. We hence combine the tests generated by the two models, GPT-3.5 and GPT-4, in post-processing to reduce the effects of self-bias.

Test Cases	Pass@1	
	GPT-4	GPT-3.5
Tests generated by GPT-4 (original)	46.28	39.69
Tests generated by GPT-3.5	41.35	38.34

Table 8: Performance of GPT-4 and GPT-3.5 under tests generated by GPT-4 or GPT-3.5. Note that all scores are computed over the subset of samples where we have tests generated by both GPT-4 and GPT-3.5.

A.3 Comparing the Code Refinement Strategy of Humans and Models

To understand why humans have substantially higher performance than models when revision is allowed, we conduct a case study to compare the strategies of humans and models when refining the code they create. Below, we present an example of the evaluation example, the code created by the human in multiple rounds, and the code generated by GPT-4. We have several observations on the difference between human-written and model-generated code.

The first observation is that **models generate more detailed and longer code in the first submission, while humans iteratively add details to the code**. The first few submissions made by humans are often short, with simple code or only calling functions defined in the context (“self.feed_forward” and “accuracy”) in this example. In contrast, models make attempts to cover all problem requirements right from their first submission. Even in cases where the problem description lacks clarity, models make assumptions and confidently output code that seems relevant to solve the problem.

The second observation is that **humans make better use of the error message** when refining unsuccessful submissions. We can see from the example that the refinements made by humans are closely related to the error message. For instance, after seeing the error message about calling the “feed_forward()” function, the participant removed the function call from the submission. In contrast, models are often “stubborn,” refusing to modify their code even after repeated occurrences of the same error. In the example below, the model keeps generating “outputs” and “targets” as the input arguments of the “monitors” function, ignoring the error message of “monitors() missing 2 required positional arguments.”

The third observation is that **humans make more “careless mistakes” than models**. We observe that human submissions sometimes involve simple syntax errors such as missing the “:” after an if statement. In contrast, models rarely make such “careless mistakes”.

To sum up, we observe that models typically generate a complete solution in one shot, while humans write code with an incremental approach.

The evaluation example

Instructions

Functionality: This function evaluates and returns the accuracy of model predictions against the set targets using the Classifier’s associated loss functions.

Input: An optional dictionary outputs.

Output: Returns a list with one tuple to monitor the accuracy of classifier predictions.

Context

```
import numpy as np

class Layer :
    def forward(self, input_data):
        raise NotImplementedError("Forward pass not implemented.")

    def backward(self, grad_output):
        raise NotImplementedError("Backward pass not implemented.")

class Loss :
    def __init__ ( self, targets = {}):
        self.targets = targets

    def accuracy ( self , outputs ) :
        pass

class Regularizer :
```



```

pass

class Classifier :
    DEFAULT_OUTPUT_ACTIVATION = 'softmax'
    OUTPUT_NDIM = 1

    def __init__ ( self , layers , loss = 'xe' , weighted = False ,
rng = 13 ) :
        self.layers = layers
        self.losses = [ Loss () ]
        self.weighted = weighted
        self.rng = np.random.default_rng(rng)

    def monitors ( self, outputs = {}, **kwargs ) :
        ...

    def feed_forward ( self , x , ** kwargs ) :
        for layer in self.layers:
            x = layer.forward(x)
        return x

```

Target code

```

return [('acc', loss.accuracy(outputs)) for loss in self.losses]

```

Test Cases

```

def test_monitors ( ) :
    model = Classifier ( [ 10 , ( 20 , 'tanh' ) , 50 ] )
    result = model.monitors ( )
    assert isinstance ( result , list ) , 'Result is not a list'
    assert len ( result ) == 1 , 'Result list length is not 1'
    assert result [ 0 ] [ 0 ] == 'acc' , 'First element of result
tuple is not "acc"'

```

The revision history of human

Human Submission 1

```
def monitors ( self, outputs = {}, **kwargs) :  
    acc_list = []  
    outputs['predictions'] = self.feed_forward(**kwargs)  
  
    for x in self.losses:  
        acc_list.append(x.accuracy(outputs))  
  
    return acc_list
```

Result: Error –

Traceback (most recent call last):

```
File "tmp/split3_test3.py", line 52, in <module>  
    test_monitors ( )  
File "tmp/split3_test3.py", line 45, in test_monitors  
    result = model.monitors ( )  
File "tmp/split3_test3.py", line 31, in monitors  
    outputs['predictions'] = self.feed_forward(**kwargs)  
TypeError: feed_forward() missing 1 required positional argument: 'x'
```

Human Submission 2

```
def monitors ( self, outputs = {}, **kwargs) :  
    acc_list = []  
  
    for x in self.losses:  
        acc_list.append(x.accuracy(outputs))  
  
    return acc_list
```

Result: Error –

Traceback (most recent call last):

```
File "tmp/split3_test3.py", line 52, in <module>  
    test_monitors ( )  
File "tmp/split3_test3.py", line 48, in test_monitors  
    assert result [ 0 ] [ 0 ] == 'acc' , 'First element of result  
tuple is not "acc"'  
TypeError: 'NoneType' object is not subscriptable"""
```

Human Submission 3

```
def monitors ( self, outputs = {}, **kwargs) :  
    acc_list = []  
  
    for x in self.losses:  
        acc_list.append( ('acc', x.accuracy(outputs)) )  
  
    return acc_list
```

Result: Successful

The revision history of GPT-4

Model Submission 1

```
def monitors ( self , outputs , targets ) :  
    accuracy_list = [ ]  
    for loss in self.losses :  
        accuracy = loss.accuracy ( outputs , targets )  
        accuracy_list.append ( ( "accuracy" , accuracy ) )  
    return accuracy_list
```

Result: Error –

```
Traceback (most recent call last):  
  File "tmp/code_gen_round0_24.py", line 71, in <module>  
    test_monitors()  
  File "tmp/code_gen_round0_24.py", line 66, in  
test_monitors  
    result = model.monitors ( )  
TypeError: monitors() missing 2 required positional arguments:  
'outputs' and 'targets'
```

Model Submission 2

Same as Submission 1

Model Submission 3

Same as Submission 1

Model Submission 4

```
def monitors ( self , outputs , targets ) :  
    accuracy_list = [ ( loss.__class__.__name__ + "_accuracy" , loss.  
        accuracy ( outputs , targets ) ) for loss in self.losses ]  
    return accuracy_list
```

Result: Error –

```
Traceback (most recent call last):  
  File "tmp/code_gen_round4_24.py", line 68, in <module>  
    test_monitors()  
  File "tmp/code_gen_round4_24.py", line 63, in  
test_monitors  
    result = model.monitors ( )  
TypeError: monitors() missing 2 required positional arguments:  
'outputs' and 'targets'
```

A.4 Logistic Regression Analysis on Factors Affecting Model Performance

In addition to the performance breakdown analysis in §6.3, in this section, we present a more rigorous analysis on the factors affecting model performance using logistic regression.

Analysis Data. We examine the Pass@1 scores of the following models:

- OpenChat-3.5-0106² Wang et al. (2024)
- DeepSeek-Coder-7B-instruct-v1.5³ Guo et al. (2024)
- Magicoder-S-DS-6.7B⁴ Wei et al. (2023)
- WizardCoder-33B-V1.1⁵ Luo et al. (2023)
- DeepSeek-Coder-33B-instruct⁶ Guo et al. (2024)
- CodeLlama-34B-Instruct⁷ Rozière et al. (2024)
- CodeLlama-70B-Instruct⁸ Rozière et al. (2024)
- speechless-codellama-34B-v2.0⁹
- GPT-3.5¹⁰
- GPT-4¹¹

We study the performance trends with respect to the following factors, where summary statistics for the above variables are reported in sections 5.1 and 5.2:

- The length of the target code (TargetLength)
- The length of the context code (ContextLength)
- The number of imports in the code (NumberImports)
- The number of variables in the target code (FocalVariablesCount)
- The number of variables in the full code (FullVariablesCount)
- The AST tree depth for the target code (FocalASTDepth)
- The AST tree depth for the full code (FullASTDepth)
- The number of calls made to the function in the target code (NumberFunctionCalls)

Because the logistic regression analysis requires binary outcome variables, we use the empirical Pass@1 score for this analysis, which is obtained by randomly sampling an output and checking whether or not it passes all test cases. Note that empirical pass@k is different from the standard pass@k metric reported in §6, which is defined as the probability of having at least one correct solution among k randomly selected outputs.

Analysis Setup. We performed a logistic regression analysis on examples in Exec-CSN to assess the relative impact each factor had on the empirical Pass@1 score of each model. Specifically, we built a logistic regression model with one instance per example per model with empirical Pass@1 as a categorical dependent measure. After filtering out examples where we encountered errors in AST parsing, we have 1,922 examples left. Thus, there were a total of 19,220 data points in the analysis.

²<https://huggingface.co/openchat/openchat-3.5-0106>

³<https://huggingface.co/deepseek-ai/deepseek-coder-7b-instruct-v1.5>

⁴<https://huggingface.co/ise-uiuc/Magicoder-S-DS-6.7B>

⁵<https://huggingface.co/WizardLM/WizardCoder-33B-V1.1>

⁶<https://huggingface.co/deepseek-ai/deepseek-coder-33b-instruct>

⁷<https://huggingface.co/codellama/CodeLlama-34b-Instruct-hf>

⁸<https://huggingface.co/codellama/CodeLlama-70b-Instruct-hf>

⁹<https://huggingface.co/uukuguy/speechless-codellama-34b-v2.0>

¹⁰<https://platform.openai.com/docs/models/gpt-3-5-turbo>

¹¹<https://platform.openai.com/docs/models/gpt-4-and-gpt-4-turbo>

The repositories represent code of different types, thus we included an indicator of repository in our analysis to account for variation across examples that is not represented in any of our covariates. There were initially 367 distinct repositories, but for the analysis, we aggregated all repositories where we had fewer than 10 examples into a single category called Other. 50 repositories had at least 10 examples. Thus, the Repository variable had 51 levels. Independent variables included Repository, Model, and the interaction between Repository and Model. In addition, we included covariates for Target Length, Context Length, NumberOfImports, FocalVariablesCount, FullVariablesCount, FocalASTDepth, FullASTDepth, and NumberFunctionCalls.

Analysis Results. The full model had AIC 22655 and BIC 26699.7. The effect of Repository on empirical Pass@1 was significant, indicating that even controlling for the factors represented by the covariates, the examples across repositories were difficult to a differential degree. The category “Other” was in the middle of the pack. The interaction between Repository and Model was not significant, thus indicating that the general performance of models relative to that of the other models did not vary significantly across repositories. The effect of FullASTDepth and NumberFunctionCalls was not significant. Thus, the interaction term and the two nonsignificant covariates were dropped from the analysis, and the model was rerun.

The full revised model had AIC 218867.7 and BIC 22405.2. All of the independent variables and covariates were significant in the revised model. TargetLength and Repository had the highest logworth for explaining variation in the dependent variable followed by model, ContextLength, NumberImports, FocalVariablesCount, FocalASTDepth, and FullVariablesCount. The significant effect of TargetLength indicates that a longer TargetLength was associated with lower empirical Pass@1. The significant effect of ContextLength indicates that lower context lengths were associated with lower empirical Pass@1. The significant effect of NumberImports indicates that more imports are associated with lower empirical Pass@1. Higher FocalVariablesCount, FocalASTDepth, and FullVariablesCount were also associated with lower empirical Pass@1.

A.5 Exec-CSN Dataset Examples

Below we show 5 representative examples from Exec-CSN. Note that we only display the original tests for brevity and readability.

In the following example, the code generation model is required to solve a problem using the `re` standard library and a provided regular expression.

Example #1: files

Instructions

Functionality: Iterates over text to find & tokenize content within `<FILENAME>` tags.

Inputs: A single string of text containing `<FILENAME>` XML-like tags.

Outputs: A generator yielding tuples of ``stream_id`` and tokenized ``tagged_doc`` within each `<FILENAME>` tag.

File Context

```
import re
from nltk.tokenize import WhitespaceTokenizer
filename_re = re.compile ( '''.*?<FILENAME docid="(P<stream_id>.*?)
">(P<tagged_doc>(.\n)*?)</FILENAME>''' )

def files ( text ) :
    ...
```

Test Cases

```
def test_files ( ) :
    sample_text_1 = '<IGNORE>This is a preamble.<FILENAME docid
="12345">Some contents here</FILENAME>And some more text'
    sample_text_2 = 'No filename tag present here at all.'
    sample_text_3 = '<FILENAME docid="67890">Another document content
</FILENAME><FILENAME docid="54321">Yet another document content</
FILENAME>'
    results_1 = list ( files ( sample_text_1 ) )
    results_2 = list ( files ( sample_text_2 ) )
    results_3 = list ( files ( sample_text_3 ) )
    assert results_1 == [ ( '12345' , 'Some contents here' ) ] , "
Test with one FILENAME tag failed."
    assert results_2 == [ ] , "Test with no FILENAME tags failed."
    assert results_3 == [ ( '67890' , 'Another document content' ) ,
( '54321' , 'Yet another document content' ) ] , "Test with
multiple FILENAME tags failed."
```

Reference Solution

```
def files(text):
    for f_match in filename_re.finditer(text):
        yield f_match.group('stream_id'), f_match.group('tagged_doc')
```

This example tests models' ability to use surrounding context to solve a task. In this case, to implement deserialization logic, models must attend to the preceding serialization logic.

Example #2: Credentials.deserialize

Instructions

Functionality: Reconstructs a Credentials instance from a serialized string.

Inputs: A `serialized_str` representing the serialized credentials string.

Outputs: A Credentials instance with user's protected resources information.

File Context

```
import json
from automatic.exceptions import CredentialsError

class Credentials :

    def __init__ ( self , ** kwargs ) :
        self.token = kwargs.get ( 'token', '' )
        self.refresh_token = kwargs.get ( 'refresh_token', '' )
        self.expiration_time = int ( kwargs.get ( 'expiration_time' ,
        0 ) )

    def serialize ( self ) :
        concatenated = '\n'.join ( [ self.token , self.refresh_token ,
        str ( self.expiration_time ) ] )
        return json.dumps ( { 'credentials' : concatenated } ,
        separators = ( ',', ':' ) )
    @ classmethod

    def deserialize ( cls , serialized_str ) :
        ...
```

Test Cases

```
def test_users ( ) :
    conduit_client = ConduitClient ( )
    assert conduit_client.users ( "PHID-USER-1" ) == json.dumps ( { "
    result" : { "PHID-USER-1" : { "phid" : "PHID-USER-1" , "userName"
    : "user1" } } } )
    assert conduit_client.users ( "PHID-INVALID" ) == json.dumps ( { "
    result" : { } } )
    assert conduit_client.users ( "PHID-USER-1" , "PHID-USER-3" ) ==
    json.dumps ( { "result" : { "PHID-USER-1" : { "phid" : "PHID-USER
    -1" , "userName" : "user1" } , "PHID-USER-3" : { "phid" : "PHID-
    USER-3" , "userName" : "user3" } } } )
```

Reference Solution

```
@classmethod
def deserialize(cls, serialized_str):
    try:
        # JSON decode.
        data = json.loads(serialized_str)
        token, refresh_token, expiration_time = data['credentials']
        ].split('\n')
```

```
        # Create a Credentials instance.
        return cls(token=token, refresh_token=refresh_token,
expiration_time=expiration_time)
    except Exception as e:
        raise CredentialsError('Failed to deserialize credentials:
{}'.format(e))
```


This example illustrates how our CodeBenchGen framework generates new functions or classes to ensure the self-consistency of the code. In this example, the input code is making HTTP requests, while the information of the website is missing. To avoid making web requests, GPT-4 alters the code to call a mock request function which returns a `MockResponse` object. We also observed other workarounds in the data, such as tests that monkey-patch problematic libraries with safer code.

Example #3: `Client.get_container`

Instructions

Functionality: Retrieve information about a specified container, potentially applying filters and settings through query parameters.

Inputs: ``container`` (required, string), ``headers`` (optional, dictionary), ``prefix`` (optional, string), ``delimiter`` (optional, string), ``marker`` (optional, string), ``end_marker`` (optional, string), ``limit`` (optional, integer), ``query`` (optional, dictionary), ``cdn`` (optional, boolean), ``decode_json`` (optional, boolean).

Outputs: An instance of ``MockResponse`` containing the status, reason, headers, and contents (either as JSON or a string based on ``decode_json``).

File Context

```
import json

class MockResponse :

    def __init__ ( self , status , reason , headers , contents ) :
        self.status = status
        self.reason = reason
        self.headers = headers
        self.contents = contents

class Client :

    def request ( self , method , path , contents , headers ,
        decode_json = False , stream = False , query = None , cdn = False
        ) :
        if method == 'GET' :
            if not cdn :
                mock_contents = json.dumps ( [ { 'name' : 'container1'
                    , 'bytes' : 1234 , 'count' : 2 } , { 'name' : 'container2' , '
                    bytes' : 5678 , 'count' : 5 } , ] )
            else :
                mock_contents = json.dumps ( { "error" : "CDN access
                not simulated" } )
            return MockResponse ( 200 if not cdn else 400 , "OK" if
            not cdn else "Bad Request" , { 'content-type' : 'application/json'
            } , mock_contents )
            return MockResponse ( 400 , "Bad Request" , { } , "" )

    def get_container ( self , container , headers = None , prefix =
        None , delimiter = None , marker = None , end_marker = None ,
        limit = None , query = None , cdn = False , decode_json = True ) :
        ...
```

Test Cases

```
def test_get_container ( ) :
    client = Client ( )
    response = client.get_container ( "my_container" )
    assert response.status == 200
    assert response.reason == "OK"
    assert isinstance ( response.contents , list ) and response.
    contents [ 0 ] [ 'name' ] == 'container1'
    response = client.get_container ( "my_container" , decode_json =
    False )
    assert isinstance ( response.contents , str )
    response_cdn = client.get_container ( "my_container" , cdn = True
    )
    assert response_cdn.status == 400
```

Reference Solution

```
def get_container(self, container, headers=None, prefix=None,
delimiter=None, marker=None, end_marker=None, limit=None, query=
None, cdn=False, decode_json=True):
    query = dict(query or {})
    query['format'] = 'json'
    if prefix:
        query['prefix'] = prefix
    if delimiter:
        query['delimiter'] = delimiter
    if marker:
        query['marker'] = marker
    if end_marker:
        query['end_marker'] = end_marker
    if limit:
        query['limit'] = limit
    response = self.request('GET', '', '', headers, decode_json=
decode_json, query=query, cdn=cdn)
    if decode_json:
        try:
            response.contents = json.loads(response.contents)
        except json.JSONDecodeError:
            pass
    return response
```

This example tests models' ability to handle longer code contexts as well as their ability to use libraries like matplotlib.

Example #4: Striplog.plot_axis

Instructions

Functionality: Render a visual representation of geological intervals on a given Matplotlib axis.

Inputs: ``ax`` (Matplotlib axis), ``legend``, ``ladder``, ``default_width``, ``match_only``, ``colour``, ``colour_function``, ``cmap``, ``default``, ``width_field``, and additional keyword arguments for patch properties.

Outputs: None (modifies the Matplotlib axis in place).

File Context

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib as mpl
from collections import UserDict
from collections import defaultdict

class Position ( UserDict ) :

    def __init__ ( self , z , * args , ** kwargs ) :
        self.z = z
        super ( ).__init__ ( * args , ** kwargs )

class Component ( UserDict ) :

    def __init__ ( self , data = { } , * args , ** kwargs ) :
        super ( ).__init__ ( data , * args , ** kwargs )

class Decor ( UserDict ) :

    def __init__ ( self , width = None , colour = 'black' , * args ,
** kwargs ) :
        self.width = width
        self.colour = colour
        super ( ).__init__ ( * args , ** kwargs )

class Interval :

    def __init__ ( self , top , base , components = [ ] , primary =
None , description = '' , data = { } , * args , ** kwargs ) :
        self.top = Position ( top , data )
        self.base = Position ( base , data )
        self.components = components
        self.primary = primary
        self.description = description
        self.data = data
        super ( ).__init__ ( * args , ** kwargs )

    def thickness ( self ) :
        return self.base.z - self.top.z

class Striplog :
```

```

def __init__ ( self , list_of_Intervals , source = None , order =
'auto' ) :
    self._list = list_of_Intervals
    self.order = order
    self.source = source
    self.__index = 0

def plot_axis ( self , ax , legend = None , ladder = False ,
default_width = 1 , match_only = None , colour = None ,
colour_function = None , cmap = None , default = None ,
width_field = None , ** kwargs ) :
    ...

def get_ylim ( self , ax , z , other = None ) :
    y = z.z
    ymax = 1
    ymin = min ( ymax , y / self.stop )
    return ymin , ymax

def axis_transform ( self , ax , x , z , data = None , other =
None , ylim = ( 0 , 1 ) ) :
    return z , x , ylim [ 0 ] , ylim [ 1 ]

@property
def start ( self ) :
    return self._list [ 0 ].top.z

@property
def stop ( self ) :
    return self._list [ - 1 ].base.z

```

Test Cases

```

def test_plot_axis ( ) :
    fig , test_ax = plt.subplots ( )
    test_intervals = [ Interval ( 1 , 2 , components = [ Component (
{ 'lithology' : 'Limestone' } ) ] , primary = Decor ( 0.5 , 'gray'
) ) , Interval ( 2 , 3 , components = [ Component ( { 'lithology'
: 'Shale' } ) ] , primary = Decor ( 1.0 , 'green' ) ) , Interval
( 3 , 5 , components = [ Component ( { 'lithology' : 'Sandstone'
} ) ] , primary = Decor ( 0.75 , 'red' ) ) ]
    test_striplot = Striplog ( test_intervals )
    test_striplot.plot_axis ( test_ax , width_field = 'primary' )
    assert len ( test_ax.patches ) == 3 , "There should be 3 patches
on the axis."
    for rect in test_ax.patches :
        assert isinstance ( rect , mpl.patches.Rectangle ) , "Each
patch should be a Rectangle."
        assert rect.get_width ( ) in [ iv.primary.width for iv in
test_striplot._list ] , " Rectangle SPACETOKEN width SPACETOKEN
should SPACETOKEN match SPACETOKEN the SPACETOKEN 'primary'
SPACETOKEN attribute SPACETOKEN of SPACETOKEN the SPACETOKEN
intervals."

```

Reference Solution

```
def plot_axis(self, ax,
               legend=None,
               ladder=False,
               default_width=1,
               match_only=None,
               colour=None,
               colour_function=None,
               cmap=None,
               default=None,
               width_field=None,
               **kwargs
               ):
    cdata = [getattr(i, width_field) for i in self._list] #
    Access using single underscore.

    for iv, c in zip(self._list, cdata): # Access using single
    underscore.
        _, ymin = self.get_ylim(ax, iv.base)
        _, ymax = self.get_ylim(ax, iv.top)
        rect = mpl.patches.Rectangle((0, iv.top.z), c.width, iv.
        thickness(), clip_on=False, **kwargs)
        ax.add_patch(rect)
        ax.axvline(c.width, ymin=ymin, ymax=ymax, clip_on=False)
```

A.6 Exec-CSN Code Generation Prompt

We provide each model with a prompt consisting of a boilerplate natural language instruction, the surrounding context, and the header and docstring of the target code (which is a function in Exec-CSN). Note that instead of using the original docstring of the function, we use the model-generated functionality-input-output instructions generated in Step 4 (post-processing) of our framework. An example prompt is shown below.

Example Exec-CSN code generation prompt

Complete the CkClass.flags2text function in the code below based on the docstring. Output one complete piece of code. Your code should start with a ```python delimiter and end with a ``` delimiter.

```
```python
from __future__ import print_function
import os
import sys

class CkClass (object) :
 flags_dict = dict ()
 fields = dict ()
 flags = 0

 def flags2text (self) :
 """
 Functionality: Converts the 'self.flags' field into a
 list of strings representing set flag bits.
 Inputs: No external inputs; uses class instance's
 'self.flags' and 'self.flags_dict'.
 Outputs: List of strings corresponding to set flags.
 """

 ...

...
```
```