REPOST: Scalable Repository-Level Coding Environment Construction with Sandbox Testing

Anonymous Authors[1]

Abstract

We introduce REPOST, a scalable method to build repository-level code generation environments that provide execution feedback for model training. Unlike existing works that require building the entire repository for execution, which is challenging for both human and LLMs and limits the scalability of the datasets, we leverage *sandbox testing*, which isolates the target function and its dependencies to a separate script for testing. In inference, models can still access the natural repository for code generation, and the script will be used to provide execution feedback. We use our method to construct REPOST-TRAIN, a large-scale train set with 7,415 functions from 824 repositories. Training with the execution feedback provided by REPOST-TRAIN leads to a performance gain of 5.5% Pass@1 on HumanEval and 3.5% Pass@1 on RepoEval.[1]

1. Introduction

Code generation is a special NLP task that can benefit from execution-based feedback (Simon, 1963; Feng et al., 2020; Chen et al., 2021). Prior work has demonstrated the effectiveness of execution-based signals for constructing training datasets (Ni et al., 2024; Zhang et al., 2024; Liu et al., 2023), particularly for script-level code generation (e.g., LeetCode problems). However, it is non-trivial to build **execution-based** datasets for **repo-level** code generation at a **large scale**, which aims to generate code for real repositories.

One major challenge is to set up executable environments. Existing repo-level datasets typically perform *integration testing* that executes test files within the repositories, which requires building the entire repository. The process is challenging for both human and LLMs, either requiring huge manual effort (Zhang et al., 2023; Jimenez et al., 2024; Pan et al., 2024) or suffering from a low success rate of converting natural repositories to coding environments when automated (Jain et al., 2024).

[1]Code and datasets available at https://anonymous.4open.science/r/RepoST-4018.

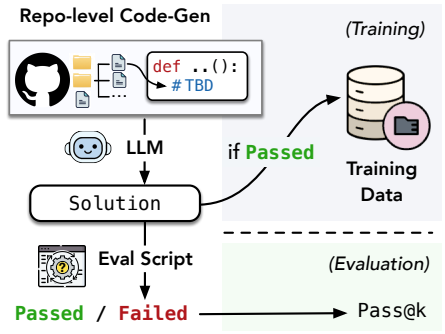


Figure 1: To build a training set, we first apply the code generation model to generate candidate solutions with the original repository as context. Then we evaluate the solutions by executing the evaluation script built by REPOST. All successful solutions are added as training targets.

In this work, we present REPOST, an **automated** framework to construct **Repo**-level coding environments with ***S**andbox **T**esting*, as an alternative to traditional *integration testing*. Given a target function within a GitHub repository, REPOST isolates the function along with its local dependencies into a separate script and generates equivalence tests to verify whether a model-generated solution exhibits the same behavior. Unlike approaches that require building the entire repository, our method only installs the external dependencies for the target function, which significantly simplifies environment setup and enhances dataset scalability. To ensure dataset quality, we conduct execution-based, AST-based, and LLM-based quality checks to harness the evaluation script's executability, functionality equivalence, test coverage, and correctness. As illustrated in Figure 1, our dataset enables models to access the natural GitHub repositories to generate candidate training outputs. We then use the evaluation script to obtain execution feedback.

We construct REPOST-TRAIN, a coding environment dataset for model training. As shown in Table 1, REPOST-TRAIN is, to our knowledge, the largest repo-level code generation dataset with execution support, containing 7,415 functions sampled from 824 repositories. Experiments demonstrate that training on REPOST-TRAIN improves performance on both algorithm problems in HumanEval (Chen et al., 2021) and repo-level tasks in RepoEval (Zhang et al.,

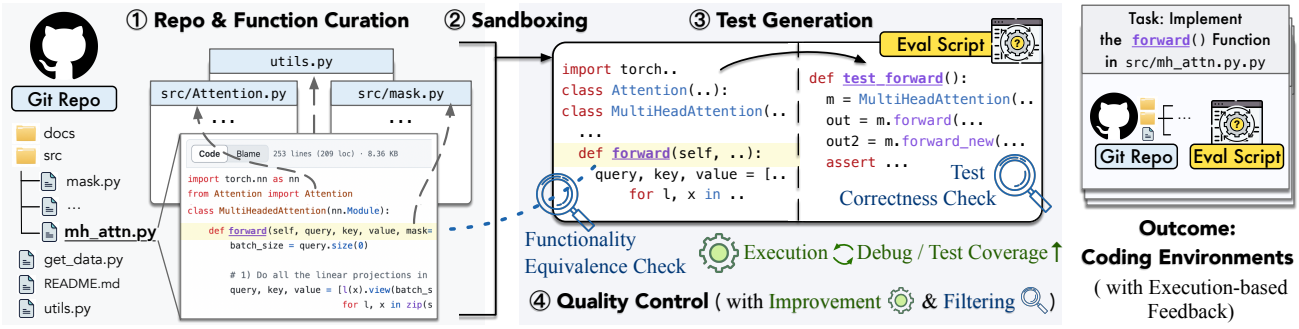


Figure 2: The REPOST coding environment construction framework. We sandbox the target function and its dependencies to a separate evaluation script, and generate tests to compare its behavior with model-generated code. The process avoids building the entire repository. We design careful quality control strategies with iterative quality improvement and post-filtering. The outcome of REPOST is a set of executable repo-level coding environments, which can be used for training.

2023). For instance, we improve Qwen2.5-Coder by 5.5% Pass@1 on HumanEval and 3.5% Pass@1 on RepoEval.

Since REPOST-TRAIN provides coding tasks with the real-world repositories as model inputs and it is possible to wrap the evaluation script in a callable API, we would like to highlight that REPOST can also be applied to coding agent (Yang et al., 2024; Wang et al., 2025) training in future works.

2. The REPOST Framework

Existing methods typically construct coding environments with **integration testing**, which creates test files that import the target function from the codebase and require the challenging process of building the entire repository. Instead, we present a framework based on **sandbox testing**. As shown in Figure 2, the key idea is to isolate the original target function and its local dependencies to a separate script, and then generate a set of equivalence tests that compare the functionality of the generated and original function.

Note that we keep a shared Docker for all the evaluation scripts. We provide all the details and statistics in §C.

2.1. Preprocessing and Evaluation Script Generation

Repository and Function Curation. We first randomly sample non-forked, MIT-licensed, Python GitHub Repositories with file sizes smaller than `10M`. Then we extract functions from the repositories. To reduce the need for computational or external resources, we follow R2E (Jain et al., 2024) and filter out functions associated with GPUs, cloud tasks, etc. by keywords.

Sandboxing: Key to Environment Setup. To isolate the target function and its necessary dependencies, we leverage the call graph to extract such local dependencies that the target function directly or indirectly calls. In principle, such context should be sufficient for the executability of the target function. Then we combine all the code fragments into the context and prompt an LLM (e.g., GPT-4o) to aggregate all the code fragments into a single script, with as little editing as possible.

To handle the challenging case where the function accesses external APIs and files, we explicitly prompt the LLM to create mock connections or mock strings if necessary. We provide an example case in Figure 4 and more examples in Figures 5 to 10. In §2.2, We will introduce the strategies to ensure that the target function's functionality does not alter.

Test Generation: Equivalence Testing. To evaluate the correctness of the generated code, we prompt the LLM to (1) generate a set of test inputs to the target function and (2) conduct equivalence testing that checks whether the generated function has the same behavior as the sandboxed original function. Compared to traditional methods that specify the I/O examples in the test cases (Chen et al., 2021), equivalence testing does not need to predict the expected outputs and is more feasible for LLMs (Jain et al., 2024). Similarly to the sandboxing step, as shown in Figures 8 to 10, we also observe that the LLM is able to generate tests with the mock classes created in the sandboxing step as context. We will check the coverage and correctness of the tests in §2.2.

2.2. Quality Control and Filtering

Executability Control with Iterative Debugging. In principle, if the model-generated function is exactly the same as the ground truth, the evaluation scripts should be successfully executed, with all the tests passed. Hence, we copy the ground truth function and execute the examples sequentially. If there are any execution errors or assertion errors, we provide the error message as the context and prompt an LLM to debug the evaluation script. Examples that still have errors after k execution-debugging iterations are filtered out.

During the process, we dynamically install external packages by reading the package names in `ModuleNotFound` and prompting the LLM to output package installation commands during debugging, if necessary.

Test Coverage Control with Iterative Improvement. After the executability control step, if the branch coverage rate is lower than some threshold (we set 80% in our experiments), we provide the LLM with the missing lines and prompt it to add tests for larger coverage.

Functionality Equivalence Check. To ensure the validity of sandbox testing, we examine the **functionality equivalence** between the sandboxed and original function. We first compare the AST of the function bodies, which is a sufficient condition of functionality equivalence. In our resulting dataset, 81.7% of the examples have the same AST. The remaining ones are filtered out from the evaluation set. To include more examples in the train set, since it is also possible that code with the same functionality has different ASTs (e.g., HTML tags can be parsed with `LexborHTMLParser` and `BeautifulSoup`), we prompt an LLM to compare the functionality equivalence, which is shown to have high agreement with human (see our human study in §3 for details).

Test Correctness Check. In principle, a test that calls the target function without any assertion checks still has a 100% coverage rate. We hence conduct **test correctness** check and apply an LLM to check whether the tests are correct, reasonable, and are completing the verification of the functionality, as listed in Table 14. Human study in §3 demonstrates the high precision of the LLM checker.

3. Resulting Dataset: REPOST-TRAIN

Statistics. We build our dataset, REPOST-TRAIN, based on repositories created between `2023-01-31` and `2024-08-31`. As shown in Table 1, to our knowledge, REPOST-TRAIN is currently the largest repo-level dataset with execution feedback. Table 3 shows detailed statistics of our datasets. We achieve on average 5.7 tests with a branch coverage rate of 97.8%. Furthermore REPOST can create relatively complex examples with on average 75.7 lines in the evaluation script, covering 894 external libraries.

Human Studies. We conduct a human study to compute the agreement between human and LLM-based functionality equivalence and test correctness checks (as introduced in §2.2). Results in Table 4 show that all 13/20 examples that pass GPT-4o's functionality check are also predicted as "same functionality" by human. Among 10 examples that pass GPT-4o's test quality check, 9 of them are predicted as "high-quality tests" by human. It demonstrates that after applying our filtering strategies for quality control, the remaining examples have high quality.

Dataset	#Examples	#Repo	Repo?	Auto Test?
HumanEval	164	–	✗	✗
DS1000	1,000	–	✗	✗
ClassEval	100	–	✗	✗
RepoEval-Func	455	6	✓	✗
SWE-Bench	2,294	12	✓	✗
CoderEval	230	43	✓	✗
DevEval	1,874	117	✓	✗
EvoCodeBench	275	25	✓	✗
SWE-Gym	2,438	11	✓	✗
R2E-Eval1	246	137	✓	✓
R2E (Our Input)	744	123	✓	✓
REPOST-TRAIN	**7,415**	**824**	✓	✓

Table 1: Statistics of REPOST-TRAIN compared to existing **execution-based** code generation datasets. R2E (Our Input) applies the R2E method to the same set of input repositories as REPOST-TRAIN, but results in a smaller number of repositories and examples. "Repo?" and "Auto Test?" refer to the repo-level setting and automatically generated tests.

As shown in Table 5, we conduct another human study to check whether the examples are reasonable and can be solved by human by sampling 27 examples generated by REPOST. Results show that 81.5% of the examples were solved by human, indicating that most examples are reasonable and not too complicated. More experimental details can be found in §D.

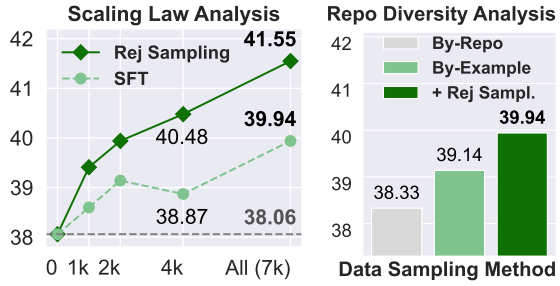
4. Code Generation Training Experiments

Training with REPOST-TRAIN. In standard supervised fine-tuning (**SFT**), we can train the model with the code context c as the input and the ground truth target function f^* as the output. The execution feedback provided by our REPOST evaluation scripts further allows us to employ the **rejection sampling fine-tuning (RFT)** algorithm to generate additional valid training targets. Specifically, we apply the model itself to our dataset, generating n candidate solutions for each function based on its code context: $(c, f_1), \ldots (c, f_n)$. The method is denoted as **RFT (Self)**. We can also apply other stronger models to generate candidates (denoted as **RFT (Distill)**). Only solutions that pass our test cases are retained. We then finetune the model on the successful functions $(c, f'_1), \ldots (c, f'_m)$ and the ground truth (c, f^*) using the standard negative log-likelihood loss.

Experimental Setup. We train two models: StarCoder2-7B (Lozhkov et al., 2024) and Qwen2.5-Coder-7B (Hui et al., 2024) with REPOST-TRAIN and evaluate on HumanEval (Chen et al., 2021), an algorithm problem dataset, and RepoEval-function (Zhang et al., 2023), a repo-level code generation dataset. For RepoEval, we use the "Oracle" context to mitigate the bias of context retrieval methods. We report the Pass@1 scores on all datasets. More training and

Model	HumanEval		RepoEval-Func	
	Pass@1	Δ	Pass@1	Δ
StarCoder2-7B	34.76	–	32.98	–
+ SFT	37.20	↑2.44	33.78	↑0.80
+ RFT (Self)	39.63	↑4.87	34.58	↑1.61
+ RFT (Distill)	**40.24**	**↑5.49**	**35.12**	**↑2.14**
Qwen2.5-Coder-7B	79.27	–	38.06	–
+ SFT	80.48	↑1.21	39.94	↑1.88
+ RFT (Self)	**84.76**	**↑5.49**	40.75	↑2.69
+ RFT (Distill)	**84.76**	**↑5.49**	**41.55**	**↑3.49**

Table 2: Code generation training results. We evaluate Pass@1 for all experiments. For RepoEval, we use the "Oracle" repo-level context as used in their original paper.



(a) Scaling law analysis (b) Repository diversity

Figure 3: (a) Pass@1 scores on RepoEval with different numbers of training examples. (b) Pass@1 scores on RepoEval with different methods to sample 2,000 training examples. Sample-by-Example has a broader repository coverage and achieves better Pass@1. The performance is further enhanced with Rejection Sampling (Distill).

evaluation details are provided in §E.

Main Results. Table 2 demonstrates that models trained with REPOST-TRAIN can generalize well to other public benchmarks. Specifically, we improve Qwen2.5-Coder by 5.5% Pass@1 on HumanEval and 3.5% on RepoEval-Func. Furthermore, in all experiments, training with RFT, even with self-training only, achieves better performance than finetuning on the original GitHub function only. For instance, RFT (Distill) outperforms SFT by 4.3% Pass@1 on HumanEval. This shows the benefit of training with environments that can provide execution feedback. We can also observe that RFT with self-training in general has lower performance than distilling from stronger models, which is probably due to fewer successful outputs (Table 6).

Scaling Law Analysis. In Figure 3(a), we investigate how the scale of training data affects model performance. Specifically, we randomly sample different numbers of examples from REPOST-TRAIN to train the model. We can see that the performance of RFT (Distill) increases as we scale up the training data, **which suggests the advantage of training data with high scalability**. Furthermore, with different scales of data, RFT consistently achieves better performance than SFT, which further demonstrates the effectiveness of training environments with execution feedback.

Repository Diversity Experiment. Figure 3(b) examines whether broader repository coverage leads to better performance, given a fixed budget of training examples. We fix the number of examples to 2,000 and experiment with two example sampling methods: (1) Sample-by-Repo, where we keep sampling repositories and adding all the functions in the repository to the training set, until the data size reaches 2,000; (2) Sample-by-Example, which is the same setting as Figure 3(a), where we randomly sample functions from REPOST-TRAIN. We observe that Sample-by-Example, covering 678 repositories, outperforms Sample-by-Repo, which only covers 221, suggesting that **training data with broader repository coverage benefits model training**.

5. Related Work

Existing works have shown the effectiveness of pretraining (Rozière et al., 2024; Lozhkov et al., 2024; Guo et al., 2024) or instruction tuning (Wei et al., 2023; Luo et al., 2023; DeepSeek-AI, 2025) on real-world code. To further finetune models for code generation, existing works have built large-scale training datasets with test cases by leveraging large-scale online algorithm problems. The execution feedback from test cases is used in constructing training targets (Ni et al., 2024; Zhang et al., 2024) or reward signals (Liu et al., 2023; Jiang et al., 2024). As repo-level code generation, existing works such as SWE-Gym (Pan et al., 2024) build training environments by manually setting up dependencies and configurations for the entire GitHub repositories. The repository setup process is complicated and challenging to automate, resulting in limited dataset scales. In comparison, we design a sandbox testing method that only requires setting up the necessary dependencies for individual GitHub functions, which reduces the difficulty of environment setup and leads to better scalability.

6. Conclusion and Future Works

We present REPOST, a scalable method to construct environments for code generation in real-world repositories that support sandbox testing. REPOST is fully automatic and enables the construction of scalable execution-based training environments. Experiments demonstrate that training with the resulting dataset, REPOST-TRAIN, leads to performance gain on other public benchmarks. For instance, we improve Qwen2.5Coder by 5.49%/3.49% Pass@1 on HumanEval/RepoEval. In future works, REPOST can also be applied to more repo-level tasks, further analysis, and coding agent training (see §A for details).

References

Bogin, B., Yang, K., Gupta, S., Richardson, K., Bransom, E., Clark, P., Sabharwal, A., and Khot, T. SUPER: Evaluating agents on setting up and executing tasks from research repositories. In Al-Onaizan, Y., Bansal, M., and Chen, Y.-N. (eds.), *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pp. 12622–12645, Miami, Florida, USA, November 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.emnlp-main.702. URL https://aclanthology.org/2024.emnlp-main.702/.

Chen, M., Tworek, J., Jun, H., Yuan, Q., de Oliveira Pinto, H. P., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., Ray, A., Puri, R., Krueger, G., Petrov, M., Khlaaf, H., Sastry, G., Mishkin, P., Chan, B., Gray, S., Ryder, N., Pavlov, M., Power, A., Kaiser, L., Bavarian, M., Winter, C., Tillet, P., Such, F. P., Cummings, D., Plappert, M., Chantzis, F., Barnes, E., Herbert-Voss, A., Guss, W. H., Nichol, A., Paino, A., Tezak, N., Tang, J., Babuschkin, I., Balaji, S., Jain, S., Saunders, W., Hesse, C., Carr, A. N., Leike, J., Achiam, J., Misra, V., Morikawa, E., Radford, A., Knight, M., Brundage, M., Murati, M., Mayer, K., Welinder, P., McGrew, B., Amodei, D., McCandlish, S., Sutskever, I., and Zaremba, W. Evaluating large language models trained on code, 2021.

DeepSeek-AI. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning, 2025. URL https://arxiv.org/abs/2501.12948.

Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, B., Liu, T., Jiang, D., and Zhou, M. 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.

Gautam, D., Garg, S., Jang, J., Sundaresan, N., and Moghaddam, R. Z. Refactorbench: Evaluating stateful reasoning in language agents through code. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=NiNIthntx7.

Guo, D., Zhu, Q., Yang, D., Xie, Z., Dong, K., Zhang, W., Chen, G., Bi, X., Wu, Y., Li, Y. K., Luo, F., Xiong, Y., and Liang, W. Deepseek-coder: When the large language model meets programming – the rise of code intelligence, 2024.

Hui, B., Yang, J., Cui, Z., Yang, J., Liu, D., Zhang, L., Liu, T., Zhang, J., Yu, B., Lu, K., Dang, K., Fan, Y., Zhang, Y., Yang, A., Men, R., Huang, F., Zheng, B., Miao, Y., Quan, S., Feng, Y., Ren, X., Ren, X., Zhou, J., and Lin, J. Qwen2.5-coder technical report, 2024. URL https://arxiv.org/abs/2409.12186.

Jain, N., Shetty, M., Zhang, T., Han, K., Sen, K., and Stoica, I. R2E: Turning any github repository into a programming agent environment. In Salakhutdinov, R., Kolter, Z., Heller, K., Weller, A., Oliver, N., Scarlett, J., and Berkenkamp, F. (eds.), *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 21196–21224. PMLR, 21–27 Jul 2024. URL https://proceedings.mlr.press/v235/jain24c.html.

Jiang, N., Li, X., Wang, S., Zhou, Q., Hossain, S. B., Ray, B., Kumar, V., Ma, X., and Deoras, A. Ledex: Training LLMs to better self-debug and explain code. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024. URL https://openreview.net/forum?id=d1XrZ4EINV.

Jimenez, C. E., Yang, J., Wettig, A., Yao, S., Pei, K., Press, O., and Narasimhan, K. R. 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.

Li, J., Li, G., Zhao, Y., Li, Y., Jin, Z., Zhu, H., Liu, H., Liu, K., Wang, L., Fang, Z., Wang, L., Ding, J., Zhang, X., Dong, Y., Zhu, Y., Gu, B., and Yang, M. Deveval: Evaluating code generation in practical software projects, 2024.

Liu, J., Zhu, Y., Xiao, K., FU, Q., Han, X., Wei, Y., and Ye, D. RLTF: Reinforcement learning from unit test feedback. *Transactions on Machine Learning Research*, 2023. ISSN 2835-8856. URL https://openreview.net/forum?id=hjYmsV6nXZ.

Lozhkov, A., Li, R., Allal, L. B., Cassano, F., Lamy-Poirier, J., Tazi, N., Tang, A., Pykhtar, D., Liu, J., Wei, Y., Liu, T., Tian, M., Kocetkov, D., Zucker, A., Belkada, Y., Wang, Z., Liu, Q., Abulkhanov, D., Paul, I., Li, Z., Li, W.-D., Risdal, M., Li, J., Zhu, J., Zhuo, T. Y., Zheltonozhskii, E., Dade, N. O. O., Yu, W., Krauß, L., Jain, N., Su, Y., He, X., Dey, M., Abati, E., Chai, Y., Muennighoff, N., Tang, X., Oblokulov, M., Akiki, C., Marone, M., Mou, C., Mishra, M., Gu, A., Hui, B., Dao, T., Zebaze, A., Dehaene, O., Patry, N., Xu, C., McAuley, J., Hu, H., Scholak, T., Paquet, S., Robinson, J., Anderson, C. J., Chapados, N., Patwary, M., Tajbakhsh, N., Jernite, Y.,

Ferrandis, C. M., Zhang, L., Hughes, S., Wolf, T., Guha, A., von Werra, L., and de Vries, H. Starcoder 2 and the stack v2: The next generation, 2024. URL https://arxiv.org/abs/2402.19173.

Luo, Z., Xu, C., Zhao, P., Sun, Q., Geng, X., Hu, W., Tao, C., Ma, J., Lin, Q., and Jiang, D. Wizardcoder: Empowering code large language models with evol-instruct, 2023. URL https://arxiv.org/abs/2306.08568.

Ni, A., Allamanis, M., Cohan, A., Deng, Y., Shi, K., Sutton, C., and Yin, P. Next: teaching large language models to reason about code execution. In *Proceedings of the 41st International Conference on Machine Learning*, ICML'24. JMLR.org, 2024.

Pan, J., Wang, X., Neubig, G., Jaitly, N., Ji, H., Suhr, A., and Zhang, Y. Training software engineering agents and verifiers with swe-gym, 2024. URL https://arxiv.org/abs/2412.21139.

Rozière, B., Gehring, J., Gloeckle, F., Sootla, S., Gat, I., Tan, X. E., Adi, Y., Liu, J., Sauvestre, R., Remez, T., Rapin, J., Kozhevnikov, A., Evtimov, I., Bitton, J., Bhatt, M., Ferrer, C. C., Grattafiori, A., Xiong, W., Défossez, A., Copet, J., Azhar, F., Touvron, H., Martin, L., Usunier, N., Scialom, T., and Synnaeve, G. Code llama: Open foundation models for code, 2024.

Simon, H. A. Experiments with a heuristic compiler. *J. ACM*, 1963. URL https://doi.org/10.1145/321186.321192.

Wang, X., Li, B., Song, Y., Xu, F. F., Tang, X., Zhuge, M., Pan, J., Song, Y., Li, B., Singh, J., Tran, H. H., Li, F., Ma, R., Zheng, M., Qian, B., Shao, Y., Muennighoff, N., Zhang, Y., Hui, B., Lin, J., Brennan, R., Peng, H., Ji, H., and Neubig, G. Openhands: An open platform for AI software developers as generalist agents. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=OJd3ayDDoF.

Wei, Y., Wang, Z., Liu, J., Ding, Y., and Zhang, L. Magicoder: Source code is all you need. *arXiv preprint arXiv:2312.02120*, 2023.

Xie, Y., Naik, A., Fried, D., and Rose, C. Data augmentation for code translation with comparable corpora and multiple references. In Bouamor, H., Pino, J., and Bali, K. (eds.), *Findings of the Association for Computational Linguistics: EMNLP 2023*, pp. 13725–13739, Singapore, December 2023. Association for Computational Linguistics. doi: 10.18653/v1/2023.findings-emnlp.917. URL https://aclanthology.org/2023.findings-emnlp.917/.

Yang, J., Jimenez, C. E., Wettig, A., Lieret, K., Yao, S., Narasimhan, K., and Press, O. Swe-agent: Agent-computer interfaces enable automated software engineering, 2024. URL https://arxiv.org/abs/2405.15793.

Zhang, D., Diao, S., Zou, X., and Peng, H. **PLUM**: Improving code lms with execution-guided on-policy preference learning driven by synthetic test cases, 2024. URL https://arxiv.org/abs/2406.06887.

Zhang, F., Chen, B., Zhang, Y., Keung, J., Liu, J., Zan, D., Mao, Y., Lou, J.-G., and Chen, W. 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.

A. Future Works

Future works may include: (1) further scaling up the datasets with more input repositories, (2) exploring the utility of different types of context in training and evaluation, (3) adapting REPOST to other repo-level coding tasks such as issue-solving (Pan et al., 2024), code translation (Xie et al., 2023), code refactoring (Gautam et al., 2025), environment setup (Bogin et al., 2024), etc., and (4) using REPOST-TRAIN to train and evaluate coding agents (Yang et al., 2024; Wang et al., 2025). This is possible because our datasets provide both access to the original GitHub repositories and execution feedback. Specifically, one can set the instruction as "generate the target function", and the coding agent will need to explore and interact with the entire repository by itself to obtain relevant information. We can then use the evaluation scripts to select successful trajectories and use them for model training.

B. Reproducibility Statement

We provide the following ways to reproduce our results: (1) We release the code for the entire REPOST pipeline, including repository and function curation, sandboxing, test generation, execution, and final-stage quality verification. (2) We release the code for training data construction, including both the SFT and RFT settings. (3) We release the REPOST-TRAIN dataset, including the repository commit ids and the evaluation scripts generated by REPOST. We also release the RFT (Distill) data constructed based on REPOST-TRAIN. (4) We release the docker image of REPOST-TRAIN, which already installs all the external packages required for executing all the evaluation scripts.

All the above resources can be found at `https://github.com/RepoST-Code-Gen-Paper/RepoST`.

C. Methodology Details

C.1. Repository and Function Curation Details

To balance the distribution of examples, we keep at most 30 functions for each repository. After this step, for the train set, we started from 1,000 repositories and obtained 17,448 functions from 851 repositories. Note that with our sandbox testing method, we do not need to build the entire repository and do not need to filter out repositories without setup files (e.g., `setup.py`). In theory, we can further scale up the dataset with more starting repositories.

C.2. Sandboxing and Test Generation Details

Sandbox Testing for External APIs and Tests. Even if all the dependencies are presented, it is still nontrivial to execute the sandboxed script if it requires external API, databases, files, etc. We explicitly prompt the LLM to create mock connections for any external API and create strings or write example files to a specific directory for file reading. Figure 4 presents a successful case of sandboxing with mock APIs. We provide examples of the generated sandboxed scripts in Figures 5 to 10.

We observe that the LLM is also able to generate tests with the mock classes created in the sandboxing step as context. As shown in Figures 8 to 10, we create mock class instances as the test inputs and still ensure that the function body of the sandboxed function remains the same as the original function.

Prompts. Table 9 shows the prompt we use to sandbox the target function and its local dependencies to a separate evaluation script. Table 10 shows the prompt template we use to generate tests in the evaluation scripts.

C.3. Quality Control Details

Prompts. Table 11 shows the prompt we use to debug the evaluation scripts if there are any errors when we copy the ground truth solution as the new implementation and execute the scripts. Table 12 shows the prompt we use to improve the coverage of the test function if there are any missing branches. Table 13 shows the prompt we use in the quality check stage, where we check whether the sandboxed and original functions have exactly the same functionality. Table 14 shows the prompt we use in the quality check stage, where we check whether the generated test function is correct, reasonable, and are completing the verification of the functionalities of the ground truth function and the new implementation.

D. Human Study Details

D.1. Quality Check Agreement Check

We ask 3 computer science graduate students to conduct functionality equivalence and test correctness checks for 20 examples built by our method (before the final filtering), with the same instruction we use to prompt the LLM checkers. The Kappa agreement scores among human annotators are 0.9179 for the functionality check and 0.7750 for the test check.

For functionality equivalence check, we randomly sample 20 examples built by our method. The instruction we present to the participants is exactly the same as the prompt for the LLM functionality checker, as shown in Table 13. Specifically, we show them the original and sandboxed functions and ask them whether their functionalities are the same. We allow minor differences including additional sanity checks or different print information.

As for the test correctness check, we randomly select 10 examples where the LLM predicts as "Yes" and 10 examples where the LLM predicts as "No". The instruction is the same as the prompt for the LLM test correctness check, as shown in Table 14. We additionally prevent checking the values of printed or logged information because it is typically difficult to match the exact same information in code generation.

D.2. Solvability Check

We assigned 27 examples constructed by REPOST to 9 computer science students, with no overlaps, and asked them to complete the function and answer questions about the difficulties of the examples.

Similar to the setting we use to benchmark coding models in §4, we show the participants the direct or indirect dependencies of the target function and ask them to complete it. After submitting an answer, the participants will see the execution results of our evaluation scripts and can choose to revise their answers accordingly or to give up. Finally, we asked them whether they used external tools (e.g., search engines) in completing the function and asked them to rate the difficulty of each example. Note that we do not directly show the evaluation script to the participants and explicitly ask them not to use any AI models.

Results show that 81.5% of the examples were solved by human, indicating that most examples are reasonable and not too complicated. The remaining examples were not solved for various reasons, such as the participant is not familiar with the task (e.g., reading html data) or related libraries (e.g., `BeautifulSoup4`), the intent of the function cannot be fully entailed from the context, etc. We also observe that the generated examples have varying complexity levels based on the usage of external tools and the difficulty ratings.

E. Code Generation Training Details

E.1. Evaluation Details

The performance of repo-level code generation also depends on the quality of the retrieved context. To mitigate the bias of retrieval models, we evaluate the models with the "Oracle" context for the two repo-level datasets. For RepoEval-Func, we follow the setting in their original paper that retrieves in-repo code fragments with the target function as the query.

E.2. Training Details

We compare three training methods: SFT, RFT (Self), and RFT (Distill). For the "Distill" method, we apply GPT-4o and Claude-3.5-Sonnet to generate and debug candidate solutions, separately, and combine their successful candidates for training. We provide the number of examples where we obtained at least one successful solution in Table 6.

To obtain more training pairs for both the "Self" and "Distill" settings, we further prompt the model itself or a stronger model to debug the failed solutions, with the error message in the context. We also train the model on successfully debugged solutions (c, f_1''), ... (c, f_k'').

We train the models with a learning rate of $2e-6$, a batch size of 32, and a warm-up ratio of 0.1 for 1 epoch. For QwenCoder, we add a linebreak after the prompt to prevent the first token of the target output from being `linebreak`.

Orig Function

```
def API_call(prompt: str) -> str:
    return API_client.generate(
        model=API_MODEL, messages=[...]
    )
```

Eval Script

```
class Mock_API(object):
  ...
  def generate(model, messages):
    return "mock_" + messages[0]['content']

API_MODEL = "model_name"
API_client = Mock_API("api_key")
def API_call(prompt: str) -> str:
    return API_client.generate(
        model=API_MODEL, messages=[...]
    )
...
```

Figure 4: An example where the LLM successfully creates a mock class, `Mock_API`, to replace real external API calls. This enables us to execute the target function `API_call`, which remains exactly the same as in the original codebase, without making real API calls.

	REPOST-TRAIN
Target Avg # Tokens (Lines)	112.4 (12.8)
Eval Script Avg # Tokens (Lines)	842.5 (75.7)
Avg # Test Cases	5.7
Avg Test Branch Coverage	97.8%
% Standalone Functions	28.1%
# External Libraries	894

Table 3: Detailed statistics of our datasets. The percentage of standalone functions (i.e., functions without local dependencies) are 28.1% and 26.4% in our datasets. Both are very close to 27%, the percentage of standalone functions among all GitHub code estimated by Li et al. (2024)

Quality Check	Functionality		Test	
Human Label (→)	Yes	No	Yes	No
GPT-4o - Yes	13/20	0	9/20	1/20
GPT-4o - No	1/20	6/20	4/20	6/20

Table 4: Agreement between human and GPT-4o on checking (1) the functionality equivalence between the sandboxed and original function, and (2) the correctness of the tests. When GPT-4o predicts "Yes" for both quality checks, it has a high agreement with human.

% Solved	% Require Tool	% Easy / Medium / Hard
81.5%	59.3%	29.6% / 51.9% / 18.5%

Table 5: Human study results for solvability verification. We ask the participants to complete the function with the same type of context we use to evaluate code generation models in §4.

Docstring (DocS)	Initial Generation	w/ Debugging
StarCoder2-7B	1327 / 7415	1573 / 7415
Qwen2.5-Coder-7B	1428 / 7415	1708 / 7415
GPT-4o	2438 / 7415	2861 / 7415
Claude-3.5	2503 / 7415	3092 / 7415
GPT-4o & Claude-3.5	**3110** / 7415	**3606** / 7415

Table 6: The number of examples where we obtained at least one successful solution in RFT (Self) and RFT (Distill). We report the number (1) after the initial round of generation, and (2) after debugging.

Sanity Checks for Sandboxing

1. The target function should exist in the evaluation script.
2. The number of tokens in the sandboxed function should NOT be more than 20 fewer than that in the original function.
3. The number of tokens in the entire evaluation script should NOT be more than 50 fewer than the number of tokens in the combination of all local dependencies.

Table 7: The sanity checks we conduct for the sandboxing step.

Sanity Checks for Test Generation

1. The target function should exist in the evaluation script.
2. The number of tokens in the sandboxed function should NOT be more than 20 fewer than that in the original function.
3. The number of tokens in the entire evaluation script should NOT be fewer than the number of tokens in the evaluation script obtained from the sandboxing step.
4. A test function named test_{func_name}() should exist.
5. The test function should call the target function.
6. The test function should call the new implementation (new_implementation_{func_name}()).
7. There should be at least 3 assertions in the test function.
8. There should be a main function.
9. The main function should call the test function.
10. If the main function calls the test function, the function call should not be in a try-except block.

Table 8: The sanity checks we conduct for the test generation step.

Sandboxing Prompt

Instructions:
- You're given a piece of PYTHON CODE containing a function called {func_name}. We also provide you the CONTEXT of the PYTHON CODE. Your goal is to aggregate the PYTHON CODE and the CONTEXT into one script, so that we can directly call the {func_name} function WITHOUT ANY MODIFICATIONS.
- You should edit the original PYTHON CODE as little as possible and you can add code only if necessary.
- DO NOT call any external API, database, etc. Instead, create a mock interface.
- Make sure that your code can be directly executed without any modification. For example, statements like 'token = "your_auth_token_here" # You need to replace this with a real token' is NOT allowed.
- If you need to write files to the disk, use '{docker_CACHE_DIR}' as the directory.

- Provide your reasoning and the revised PYTHON CODE below SOLUTION.

PYTHON CODE:
```python
{code}
```

CONTEXT:
{context}

Your answer should follow the format below:

Reasoning: ...
```python
# Your Code.
```

Do NOT include other formatting. Output every token of the content with no omission or abbreviation. For example, abbreviation like '... # the code keeps unchanged' is NOT allowed.

SOLUTION:

Table 9: The prompt we use to sandbox the target function and its local dependencies to a separate evaluation script.

Test Generation Prompt

Instructions:
- You're given a piece of PYTHON CODE containing a function called {func_name}. Assume we will later have another implentation of the {func_name} function called {func_name}_new_implementation.
- Your goal is to add (1) a test function called {test_func_name} to check whether {func_name}_new_implementation has the same functionality as the {func_name} function, and (2) a __main__ function that calls the test function.
- If the PYTHON CODE already contains a __main__ function, remove it and write a new __main__ function.
- The test function {test_func_name} should contain at least 3 assert statements. If {func_name}_new_implementation has different functionality as {func_name}, an Assertion Error should be triggered.
- The test function {test_func_name} should cover all the major branches of the {func_name} function
- DO NOT test on error handling and DO NOT test on the print information in the function.
- The __main__ function should NOT contain a try-except structure. If the implementation is incorrect, the program should have a non-zero exit code.
- You should edit the original PYTHON CODE as little as possible.
- If you need to write files to the disk, use '{docker_CACHE_DIR}' as the directory.

- Provide your reasoning and the new PYTHON CODE containing your test function {test_func_name} and the __main__ function below SOLUTION.

PYTHON CODE:
```python
{code}
```

Your answer should follow the format below:

Reasoning: ...
```python
# The new PYTHON CODE containing your test function {test_func_name} and the __main__ function.
```

Do NOT include other formatting. Output every token of your edited PYTHON CODE with no omission or abbreviation.

SOLUTION:

Table 10: The prompt we use to generate tests in the evaluation scripts.

Debugging Prompt *(for data construction)*

Instructions:
- You're given a piece of PYTHON CODE containing a function called {func_name} and its test function called {test_func_name}. Assume we will later add another function called {func_name}_new_implementation, the test function aims to check whether {func_name}_new_implementation has the same functionality as {func_name}.
- In our experiments, we implemented {func_name}_new_implementation exactly the same as {func_name}, but the PYTHON CODE cannot be successfully executed.
- Your task is to debug PYTHON CODE based on the ERROR MESSAGE.
- You should modify the code as little as possible, especially the test_{func_name} function and the {func_name} function.
- Make sure that after debugging, the test function test_{func_name} still have at least three assert statements and cover all the major branches of the {func_name} function.
- DO NOT test the logging information of error handling and DO NOT test on the print information in the function.
- If you need to write files to the disk, use '{docker_CACHE_DIR}' as the directory.

- Provide your reasoning and the debugged PYTHON CODE below SOLUTION. If necessary, output the bash scripts for Linux in another code block in the format of "'bash ... "'.

PYTHON CODE:
"'python
{code}
"'

ERROR MESSAGE:
"'
{err_msg}
"'

Your answer should follow the format below:

Reasoning: ...
"'python
The debugged PYTHON CODE in one piece.
"'
"'bash
the bash script, if necessary
"'

Do NOT include other formatting. Output every token of your debugged PYTHON CODE with no omission or abbreviation.

SOLUTION:

Table 11: The prompt we use to debug the evaluation scripts if there are any errors when we copy the ground truth solution as the new implementation and execute the scripts.

Test Coverage Improvement Prompt

Instructions:
- You're given a piece of PYTHON CODE containing a function called {func_name} and its test function called {test_func_name}. Assume we will later add another function called {func_name}_new_implementation, the test function aims to check whether {func_name}_new_implementation has the same functionality as {func_name}.
- You're also given the MISSING LINES of the {func_name}_new_implementation function that are NOT covered by {test_func_name}.
- Your task is to improve the branch coverage rate of the {test_func_name} function.
- You should only modify the {test_func_name} function. DO NOT modify other parts of the code.
- DO NOT test the logging information of error handling and DO NOT test on the print information in the function.
- If you need to write files to the disk, use '{docker_CACHE_DIR}' as the directory.

- Provide your reasoning and your revised {test_func_name} function below SOLUTION.

PYTHON CODE:
```python
{code}
```

MISSING LINES:
{missing_code}

Your answer should follow the format below:

Reasoning: ...
```python
# Your revised {test_func_name} function
```

Do NOT include other formatting. Output every token of the {test_func_name} function with no omission or abbreviation.

SOLUTION:

Table 12: The prompt we use to improve the coverage of the test function if there are any missing branches.

Functionality Equivalence Check Prompt

Instructions:
- We revised a python function called {func_name} so it can be directly executed in an isolated environment.
- You are given the ORIGINAL FUNCTION and the CODE containing the REVISED FUNCTION.
- Your task is to check whether the functionality of the REVISED FUNCTION is the same as the ORIGINAL FUNCTION.
- If the REVISED FUNCTION is exactly the same as the ORINIGAL FUNCTION, output "same" as your answer.
- Otherwise, if the functionality of the REVISED FUNCTION is the same as the ORIGINAL FUNCTION, output "yes" as your answer.
- if the functionality of the REVISED FUNCTION is different, output "no".

- Provide your reasoning and the answer under "SOLUTION".

ORIGINAL FUNCTION:
{orig_func}

CODE containing the REVISED FUNCTION:
{new_code}

Your answer should follow the format below:
```
REASONING: Your reasoning,
ANSWER: "same", "yes" or "no".
```

Do NOT include other formatting.

SOLUTION:

Table 13: The prompt we use in the quality check stage, where we check whether the sandboxed and original functions have exactly the same functionality.

Test Correctness Check Prompt

Instructions:
- You are given a piece of PYTHON CODE containing a function called {func_name}, its new implementation {func_name}_new_implementation (now hidden) and its test function called {test_func_name}.
- Your task is to judge whether the test function satisfies all the CONDITIONS:
** CONDITION 1 ** The {func_name} function should either have return values or modifies global variables or input arguments (such as a list, a dictionary, a class, etc.).
** CONDITION 2 ** The test cases should only check the return values or variable states. It should NOT check printed or logged contents.
** CONDITION 3 ** {func_name}_new_implementation can pass all the test cases IF AND ONLY IF it has the EXACTLY same functionality as {func_name}.
** CONDITION 4 ** The test cases and assert statements are reasonable. For example, if {func_name} does not have return values, you should NOT use 'assert {func_name}() == {func_name}_new_implementation()' to test the implementation.
** CONDITION 5 ** The test cases are non-trivial.

- If the test function satisfies all the CONDITIONS, answer "yes". Otherwise, answer "no".
- Provide your reasoning and the answer under "SOLUTION".

PYTHON CODE:
{code}

Your answer should follow the format below:
```
REASONING: Your reasoning,
ANSWER: "yes" or "no".
```

Do NOT include other formatting.

SOLUTION:

Table 14: The prompt we use in the quality check stage, where we check whether the generated test function is correct, reasonable, and actually comparing the functionalities of the ground truth function and the new implementation.

[Case 1, Origifinal Function & GitHub Context] `score_explicit_question()`

````
```python
## chan_questions.py
import json
import prompts

from helpers import get_openai_answer, chunker, clean_and_hash, clean_html, query_to_search_url

def score_explicit_question(string: str) -> list:
    """
    Uses LLMs to score a question based on whether it is considered explicit or implicit.
    Uses OpenAI.
    """
    prompt = prompts.IS_EXPLICIT
    answer = get_openai_answer(prompt.replace("[input]", string))

    results = json.loads(answer)["results"]
    return results
```

```python
## helpers.py
import openai
import config

def get_openai_answer(prompt: str, response_format="json_object", model=None):
    # initiate
    client = openai.OpenAI(api_key=config.OPENAI_KEY)

    if not model:
        model = config.MODEL

    # Get response
    response = client.chat.completions.create(
        model=model,
        temperature=config.TEMPERATURE,
        max_tokens=config.MAX_OUTPUT_TOKENS,
        response_format={"type": response_format},
        messages=[{
            "role": "user",
            "content": prompt
        }]
    )

    return response.choices[0].message.content
```
````

Figure 5: Case study 1. The original `score_explicit_question` function and its context extracted from the original GitHub repository. The function calls the text completion function from the OpenAI API.

[Case 1, Evaluation Script - Sandboxing] `score_explicit_question()`

```
import json

# Mock configuration
class config:
    OPENAI_KEY = "mock_openai_key"
    MODEL = "mock_model"
    TEMPERATURE = 0.5
    MAX_OUTPUT_TOKENS = 100

# Mock prompts
class prompts:
    IS_EXPLICIT = "Is the following question explicit or implicit? [input]"

# Mock OpenAI API response
def get_openai_answer(prompt: str, response_format="json_object", model=None):
    # Mock response simulating the OpenAI API
    mock_response = {
        "results": [
            {"question": prompt, "score": "explicit" if "explicit" in prompt else "implicit"}
        ]
    }
    return json.dumps(mock_response)

# The original function from chan_questions.py
def score_explicit_question(string: str) -> list:
    """
    Uses LLMs to score a question based on whether it is considered explicit or implicit.
    Uses OpenAI.
    """
    prompt = prompts.IS_EXPLICIT
    answer = get_openai_answer(prompt.replace("[input]", string))

    results = json.loads(answer)["results"]
    return results
```

Figure 6: Case study 1. The sandboxed `score_explicit_question` function in the evaluation script. The LLM generates a mock function called `get_openai_answer` to replace the real API call. With the mock class, the `score_explicit_question` has the exactly same functionality as the original function, but does not make real OpenAI API calls.

[Case 1, Evaluation Script - Tests] `score_explicit_question()`

```
def test_score_explicit_question():
    # Test case 1: Explicit question
    input_str1 = "Is this an explicit question?"
    assert score_explicit_question(input_str1) == score_explicit_question_new_implementation(input_str1)

    # Test case 2: Implicit question
    input_str2 = "What is the meaning of life?"
    assert score_explicit_question(input_str2) == score_explicit_question_new_implementation(input_str2)

    # Test case 3: Neutral question (contains neither explicit nor implicit)
    input_str3 = "How are you?"
    assert score_explicit_question(input_str3) == score_explicit_question_new_implementation(input_str3)

    # Test case 4: Very short question
    input_str4 = "?"
    assert score_explicit_question(input_str4) == score_explicit_question_new_implementation(input_str4)

    # Test case 5: Very long question
    input_str5 = "Is this an explicit question?" * 100
    assert score_explicit_question(input_str5) == score_explicit_question_new_implementation(input_str5)

    # Test case 6: Case sensitivity
    input_str6 = "is this an explicit question?"
    assert score_explicit_question(input_str6) == score_explicit_question_new_implementation(input_str6)

    # Test case 7: Different punctuation
    input_str7 = "Is this an explicit question!"
    assert score_explicit_question(input_str7) == score_explicit_question_new_implementation(input_str7)

    # Test case 8: Special characters
    input_str8 = "Is this an explicit question? #$%"
    assert score_explicit_question(input_str8) == score_explicit_question_new_implementation(input_str8)

    # Test case 9: Numbers in question
    input_str9 = "Is 42 an explicit number?"
    assert score_explicit_question(input_str9) == score_explicit_question_new_implementation(input_str9)

    # Test case 10: Empty string
    input_str10 = ""
    assert score_explicit_question(input_str10) == score_explicit_question_new_implementation(input_str10)

    # Test case 11: Leading and trailing whitespace
    input_str11 = "   Is this an explicit question?   "
    assert score_explicit_question(input_str11) == score_explicit_question_new_implementation(input_str11)

    # Test case 12: Excessive internal whitespace
    input_str12 = "Is    this    an    explicit    question?"
    assert score_explicit_question(input_str12) == score_explicit_question_new_implementation(input_str12)

# Main function
if __name__ == "__main__":
    test_score_explicit_question()
```

Figure 7: Case study 1. The tests in the evaluation scripts.

[Case 2, Origifinal Function & GitHub Context] `QuizBrain.next_question()`

````
```python
## Complete Python Bootcamp/Day 017 Quiz Game/quiz_brain.py
class QuizBrain:
    def __init__(self, questions):
        self.question_number = 0
        self.question_list = questions
        self.score = 0

    def still_has_question(self):
        return self.question_number < len(self.question_list)

    def next_question(self):
        user_answer = ""
        current_question = self.question_list[self.question_number]
        while user_answer.lower() not in ("true", "false"):
            user_answer = ""
            user_answer = input(f"Q{self.question_number + 1}: {current_question.question} (True/False)")

        self.question_number += 1
        self.check_answer(user_answer, current_question.answer)

    def check_answer(self, user_answer, correct_answer):
        if user_answer.lower() == correct_answer.lower():
            print("Correct!")
            self.score += 1
        else:
            print("Incorrect")
        print(f"The correct answer is: {correct_answer}\nYour current score is {self.score}/{self.question_number}")

```
````

Figure 8: Case study 2. The original `next_question` function. The function reads from system inputs, which are not available when testing in a docker.

[Case 2, Evaluation Script - Sandboxing] `QuizBrain.next_question()`

```
class MockInput:
    def __init__(self, responses):
        self.responses = responses
        self.index = 0

    def __call__(self, prompt):
        if self.index < len(self.responses):
            response = self.responses[self.index]
            self.index += 1
            print(prompt + response)  # Simulate user seeing the prompt and typing the response
            return response
        raise Exception("No more mock inputs available.")

class Question:
    def __init__(self, question, answer):
        self.question = question
        self.answer = answer

class QuizBrain:
    def __init__(self, questions):
        self.question_number = 0
        self.question_list = questions
        self.score = 0

    def still_has_question(self):
        return self.question_number < len(self.question_list)

    def next_question(self):
        user_answer = ""
        current_question = self.question_list[self.question_number]
        while user_answer.lower() not in ("true", "false"):
            user_answer = ""
            user_answer = input(f"Q{self.question_number + 1}: {current_question.question} (True/False)")

        self.question_number += 1
        self.check_answer(user_answer, current_question.answer)


    def check_answer(self, user_answer, correct_answer):
        if user_answer.lower() == correct_answer.lower():
            print("Correct!")
            self.score += 1
        else:
            print("Incorrect")
        print(f"The correct answer is: {correct_answer}\nYour current score is {self.score}/{self.question_number}")
```

Figure 9: Case study 2. The sandboxed `next_question` function in the evaluation script. The LLM generates a mock class called `MockInput` to replace the real system input. With the mock class, the `next_question` has the exactly same functionality as the original function, but does not read system inputs.

[Case 2, Evaluation Script - Tests] **`QuizBrain.next_question()`**

```
def test_next_question():
    questions = [
        Question("Is the sky blue?", "True"),
        Question("Is the grass red?", "False"),
    ]
    quiz_original = QuizBrain(questions)
    quiz_new = QuizBrain(questions)

    mock_responses = ["True", "False"]
    mock_input = MockInput(mock_responses)

    # Replace the built-in input function with mock_input for testing
    global input
    original_input = input
    input = mock_input

    # Run original implementation
    while quiz_original.still_has_question():
        quiz_original.next_question()

    # Reset input for the new implementation
    mock_input = MockInput(mock_responses)
    input = mock_input

    # Run new implementation
    while quiz_new.still_has_question():
        quiz_new.next_question_new_implementation()

    assert quiz_original.score == quiz_new.score, "Scores differ between implementations"
    assert quiz_original.question_number == quiz_new.question_number, "Question numbers differ between
implementations"
    assert quiz_original.still_has_question() == quiz_new.still_has_question(), "Question completion state differs
between implementations"


if __name__ == "__main__":
    test_next_question()
```

Figure 10: Case study 2. The tests in the evaluation scripts, which call the `MockInput` class to mock the system input.