Can Language Models Be Used for Code Migration?

Anonymous ACL submission

Abstract

Large language models (LLMs) have demonstrated remarkable proficiency in handling a wide range of tasks within the software engineering domain, but their ability to perform code migration—adapting code to different environments—remains underexplored. In this work, we propose a novel benchmark, CODEMENV: **Code** **M**igration Across **Env**ironment, designed to evaluate LLMs' performance in handling code migration tasks. The benchmark comprises 922 data points across 19 Python and Java packages, offering three tasks to systematically evaluate code migration: identifying version-incompatible functions, determining function changes, and adapting code to target environments. Experimental evaluation of CODEMENV across seven LLMs revealed an average pass@1 rate of 26.50%, with GPT-4O performing best at 43.84%. We highlight our key findings as follows: (i) LLMs are more familiar with newer function versions, making them better at migrating legacy code, and (ii) a logical inconsistency where LLMs sometimes identify irrelevant function changes for the target migration environment.

1 Introduction

Large Language Models (LLMs) have shown impressive capabilities in tasks such as code generation (Jiang et al., 2024; Du et al., 2024) and code translation (Yuan et al., 2024; Eniser et al., 2024). For example, state-of-the-art models such as GPT-4 (Achiam et al., 2023), Claude-3 (The), and DeepSeek (Shao et al., 2024) have demonstrated exceptional performance across various benchmarks, significantly outperforming traditional methods.

However, whether LLMs can effectively handle code migration—adapting code from one environment to another—remains an underexplored question. Code migration is essential in many scenarios. For instance, when running code from external sources, such as a GitHub repository, users often spend significant time configuring a compatible environment. If LLMs could assist in seamlessly migrating code to an existing setup, they could greatly reduce the manual effort required for environment configuration and compatibility adjustments. A key reason for code incompatibility across different environments is the challenge posed by evolving library structures, *i.e.,* continuous maintenance and updates of the library functions. For example, as shown in Figure 1, the method call for the function `compare_chararrays` has been updated from the `numpy` package to the `numpy.char` package. Consequently, the function call differs between NumPy 1.26 and NumPy 2.0.

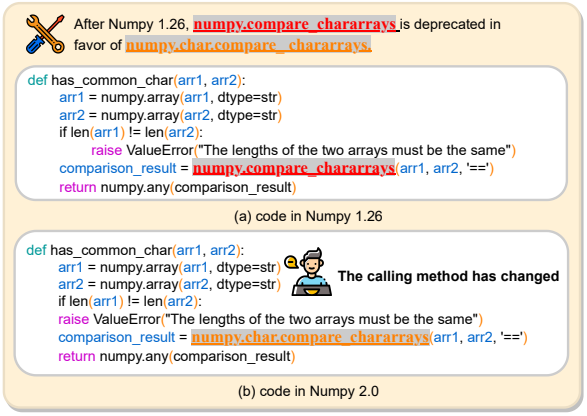


Figure 1: The function `compare_chararrays` was updated after Python version 1.26, altering its calling method, emphasizing code adjustments for compatibility.

Research in this area is still in its early stages, with only a few studies exploring the challenges and potential solutions. Some benchmarks, like CodeUpdate (Liu et al., 2024), focus on how to inject knowledge of new API functions that the model has never seen before. However, the functions used in these studies are synthetically generated by GPT rather than sourced from real-world

libraries. This limitation makes it challenging to assess the feasibility of code migration in real-world scenarios.

To bridge this gap, we propose a novel benchmark. *i.e.,* CODEMENV: **Code** **M**igration Across **Env**ironment. CODEMENV is constructed based on manually collected official function changes, with a total of 922 data points, involving Python and Java, across 19 packages. As illustrated in Figure 2, it comprises three tasks designed to evaluate the ability of different models to perform code migration, detailed as follows:

Task-1: Locate version-incompatible function. Given a piece of code and a target execution environment, the model must identify functions or code segments that may become incompatible across different versions.

Task-2: Answering function changes. The model must describe the specific changes these functions have undergone across different versions.

Task-3: Code migration. Finally, the model is required to modify the given code to ensure compatibility with the target environment by addressing the identified version incompatibilities.

For evaluation of CODEMENV, we experiment using seven different LLMs, with experimental results highlighting: The average pass@1 rate for the migration task across seven models is 26.50%. Among them, GPT-4O stands out, achieving the highest performance with an average pass@1 rate of 43.84%. Our experimental analyses reveal following interesting findings:

***(i)* Familiarity with functions**, LLMs exhibit a stronger familiarity with newer function versions than older ones, making them more effective at migrating legacy code to modern environments but less proficient at adapting newer code to older settings.

***(ii)* Logical Inconsistency**, LLMs exhibit a logical inconsistency between function change identification and migration requirements. For example, when migrating from version 1.16 to 2.0, the model answering a function change from version 1.0, which is irrelevant to the target migration environment.

2 Related Work

2.1 Large Language Models (LLMs)

Large language models, with their vast parameters and training on extensive corpora, have showcased impressive capabilities in code generation, translation, and completion. Proprietary models such as GPT-4 (Achiam et al., 2023), Claude-3 (The), and Gemini (Reid et al., 2024) have demonstrated exceptional performance in assisting users with a wide range of programming tasks. Additionally, open-source models like Qwen2.5-Coder (Team, 2024) have outperformed many models with significantly more parameters by leveraging synthetic data during training. Other open-source models like Llama-3.1 (Abhimanyu Dubey et al., 2024), Phi-3 (Abdin et al., 2024) also perform well, with DeepSeek (Shao et al., 2024) even surpassing most proprietary models.

In this paepr, we evaluate the knowledge of these large models about function changes and their ability to migrate code.

2.2 Knowledge Editing

The research on knowledge editing aims to efficiently modify model parameters to update its knowledge. Most studies in this field focus on editing natural language knowledge. ROME (Meng et al., 2022a) and MEMIT (Meng et al., 2022b) adopt a locate-then-edit paradigm, where the parameter position of the knowledge is first located, and then the parameter is updated to modify the model's knowledge. Some work (Zhong et al., 2023; Cheng et al., 2024) adopts plan-and-solve paradigm, where complex problems are decomposed into the knowledge required for each step, which are then solved one by one.

There is less research on changes to function: CodeUpdateArena (Liu et al., 2024) introduces a benchmark for updating LLMs with new API function knowledge to solve program synthesis tasks. CLMEEval (Li et al., 2024) propose a benchmark for evaluating model editing techniques on LLMs4Code, and proposes A-GRACE, an enhanced method for better generalization in code knowledge correction. Some work (Zhou et al., 2022; Su et al., 2024; Hsieh et al., 2023) uses retrieval-augmented approaches (Lewis et al., 2020; Guu et al., 2020) to provide models with code change knowledge for improving code generation.

CODEMENV does not provide the model with knowledge of function changes within the context during evaluation. Instead, we focus more on evaluating how well the model utilizes its own knowledge of function changes to perform code migration.

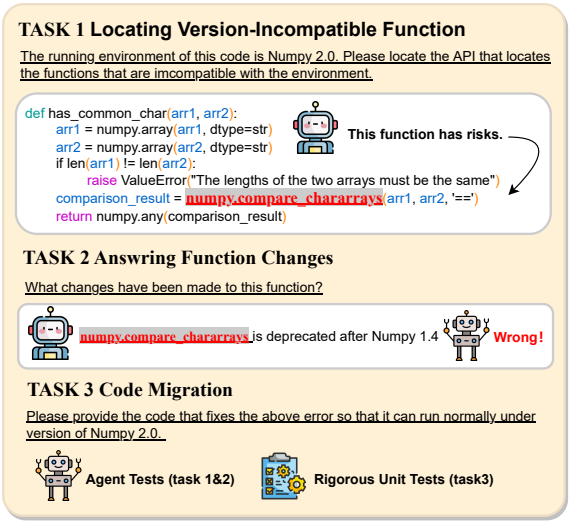


Figure 2: A data example of CODEMENV, which includes three tasks to evaluate LLMs on environment-related programming skills.

3 CODEMENV

Despite the challenges environmental issues pose for programmers, there is a lack of systematic evaluation of model capabilities in code migration across different environments. To address this gap, we propose CODEMENV: **Code** **M**igration Across **Env**ironments. CODEMENV assesses a model's understanding of function usage differences across versions and its ability to perform cross-version code migration. This section provides a detailed introduction to CODEMENV.

3.1 Task Definition

CODEMENV use three tasks to comprehensively evaluate the model's capabilities to perform code migration.

Task-1: Locate version-incompatible function. The first task presents a piece of code along with a target environment version. The model must identify functions that are incompatible with the specified environment. CODEMENV includes two difficulty levels: **easy**, featuring a single incompatible function, and **hard**, involving multiple incompatible functions.

Task-2 Answering function changes. This task requires the model to output version-related changes for the identified incompatible function. Specifically, it should determine how the function has evolved across versions, such as deprecation, parameter modifications, or replacement by a new function.

Task-3 Code Migration. This task requires the model to adjust the given code to ensure compatibility with the target environment. Code migration scenarios fall into two categories:

(a) NEW2OLD. The target environment version is lower than the original, requiring adaptation of newer code to run in a legacy environment.

(b) OLD2NEW. The original environment version is lower than the target environment version. This scenario involves upgrading older code to be compatible with a newer environment.

3.2 Dataset Statistics

CODEMENV includes two programming languages: Python and Java. The Python dataset consists of 11 packages with a total of 587 data points, categorized into two difficulty levels: easy and hard. The easy category contains 396 data points, where only a single line of code is incompatible with the target environment. The hard category contains 191 data points, where there are k lines of incompatible code $k \in \{2, 3\}$. The Java dataset consists of 8 packages with 335 data points. The Java dataset only contains easy difficulty because we find that its incompatible functions have poor linkage and it is difficult to make difficult data. See Appendix B for the details of data statistics.

3.3 Function Changes

We divide function changes into three types:

- Addition (`None` $\rightarrow f$): A new function f is introduced in a later version, meaning older environments cannot use it
- Deprecation ($f \rightarrow$ `None`): The function f is no longer supported after a certain version, making it unusable in newer environments
- Replacement ($f \rightarrow f'$): The functions f has been modified to f', with changes such as alterations to the calling method, the number of function parameters, and other adjustments.

See Table 2 for the distribution of the types of changes we collected.

3.4 Evaluation

In this section, we introduce some details of our evaluation. We mainly use the following two evaluation methods.

Agent-based Evaluation. To verify whether the LLM correctly identifies version-incompatible

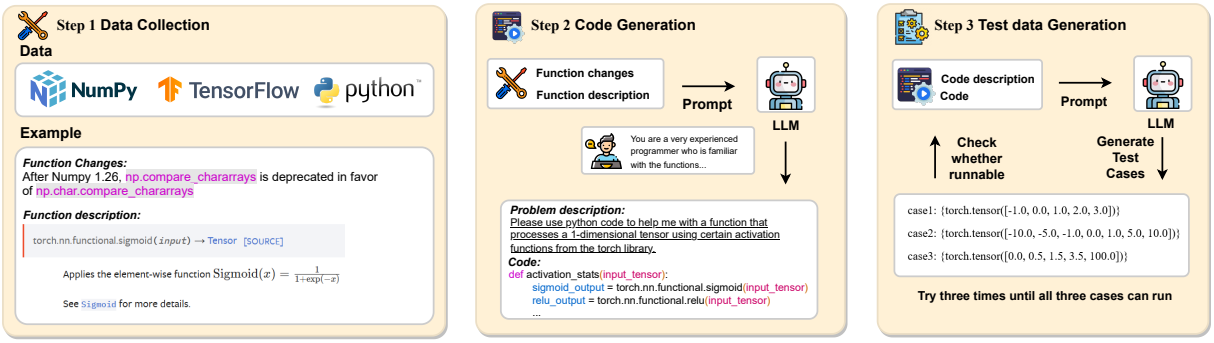


Figure 3: The construction process of CODEMENV. **Step 1:** We collect function change information and function descriptions from the official website; **Step 2:** Based on the collected functions, generate code that can run in the original version and its problem description; **Step 3:** Generate 3 test cases for each data and repeat three times until all cases can run correctly.

functions and whether the change knowledge provided for these functions is accurate, we use an agent-based evaluation approach. We provide each agent with the correct answers, including the incompatible functions and the changes these functions underwent. The agent's task is to evaluate whether the model's answers are correct. It does this by comparing the model's output with the correct answers based on the evaluation criteria we provide.

For the first task, we require the correct identification of all functions that are incompatible with the environment. Missing even one is considered a failure. For the second task, we mainly consider three aspects: First, whether the model can correctly identify the type of change, i.e., deprecated, added, or replaced. Next, whether the model can accurately identify what the function has been replaced by, skipping this step for deprecated and added functions. Finally, the third aspect is whether the model can accurately provide the version number in which the change occurred. A difference of less than 0.5 in version numbers is considered correct.

Unit Tests. To verify the correctness of the migrated code, we prepare three test cases for each data. These test cases ensure that the modified code not only eliminates environment-related issues but also preserves its original functionality. We ensure correctness by comparing the answers obtained from executing the migrated code in the target environment with the answers in the test cases. The code is considered correct if all three test cases pass.

See Appendix A for the details of prompts for evaluation.

Datasets	1-incom.	2-incom.	3-incom.	Total
Easy(Py.)	396	-	-	396
Hard(Py.)	-	103	88	191
Java	335	-	-	335

Table 1: Statistics of incompatible functions of CODEMENV .

Package	Replacement	Deprecation	Addition
numpy	2	8	-
pandas	-	12	13
tensorflow	87	2	2
python	9	7	7
math	-	1	17
re	-	-	2
os	-	-	14
random	-	-	2
csv	-	-	1
itertools	-	-	5
torch	-	5	5
total	98	35	79

Table 2: Statistics of change types collected by different packages.

3.5 Process-flow

Figure 3 illustrates the process of constructing our dataset, which consists of three main steps.

Step 1 Data Collection. Our first step is to collect a set of functions along with their changes, functional descriptions, and supported version ranges.

To achieve this, we identify which functions have changed by reviewing the version release notes on the official website of each package. Here, we can determine in which version these functions were modified and what changes were made. At the same time, we record the functional

descriptions and usage of these functions, making it easier to generate code based on them later. Official documentations do not always specify the version ranges in which these functions are compatible, thus we determine them through manual execution and verification.

Through our analysis, we identified a total of 212 functions for Python datasets and 114 functions for Java datasets, which provide essential data support for the subsequent code generation process. The websites from which we collected data are organized in Appendix.

Step 2 Code Generation. The core of our approach in this step is to provide the powerful model GPT-4 with functions and their usage methods, allowing it to generate code that calls these functions.

Depending on the type of change, we will determine whether to generate the OLD2NEW or NEW2OLD scenario. For deprecation ($f \rightarrow None$), we provide the model with the function f before it was deprecated and ask it to generate the code. The target environment version is then selected as the version after the change, where the function f can no longer be used (OLD2NEW Scenario). For addition ($None \rightarrow f$), we provide the model with the function f that was newly introduced and ask it to generate the code. The target environment version is selected as the version before the change, where the function is unavailable (NEW2OLD Scenario). For replacement ($f \rightarrow f'$), a similar approach to the previous steps can be used to generate two sets of data for the functions before and after the change, corresponding to the OLD2NEW and NEW2OLD scenarios.

In addition to generating code, we also require the model to generate the problem and input range corresponding to this code. These elements are crucial for the third step, where we generate test cases.

Step 3 Test Cases Generation. In this step, we construct the test cases. We provide GPT-4 with the code, problem description, and input range. A total of three test cases are generated. These test cases are then used to verify the correctness of the code execution and obtain the output results. However, we find that even though we provide the model with the correct input data range, the generated input data might be problematic. Therefore, we provide the compilation information error as feedback to the model to fix the test cases. Repeat this step three times until all three cases pass, otherwise discard the code. For the Python dataset of easy difficulty, we generated 629 code samples in the previous step, with 396 remaining after filtering. For the hard-difficulty Python dataset, 441 code samples were generated, leaving 191 after filtering.

4 Experimentation

In this section, we introduce the experimental settings and the analysis results of our experiments.

4.1 Experimental Settings

Large Models. We conduct experiments on seven LLMs, including three proprietary models:GPT-TURBO-3.5 (Ye et al., 2023), GPT-4O-MINI (OpenAI et al., 2024a), GPT-4O (OpenAI et al., 2024b), and four open-source models: LLAMA-3.1-8B-INSTRUCT (Abhimanyu Dubey et al., 2024), LLAMA-3.1-70B (Abhimanyu Dubey et al., 2024), QWEN2.5-CODER-7B-INSTRUCT (Team, 2024) and DEEPSEEK-R1 (Shao et al., 2024).

Evaluation Metrics. we use Pass@k (Hendrycks et al., 2021) to evaluate the effectiveness of different models. Pass@k refers to the probability that at least one correct solution is included among the top k generated solutions for each problem:

$$\text{Pass@}k := \mathbb{E}\left[1 - \frac{\binom{n-c}{k}}{\binom{n}{k}}\right] \quad (1)$$

where n is the number of coding solutions, c is the number of passed solutions, and k is the number of solutions being evaluated. For task-1 and task-2, we set $k = 1$. For task-3, we set $k \in \{1, 5\}$.

Experiment Setup. For all LLMs, we set the generation temperature to 0.7, the maximum generated length is 2048 tokens. For proprietary models such as the GPT series, we conduct evaluations using the APIs provided on their official website. For smaller open-source models, we deploy them locally with two RTX 4090 GPUs. For larger open-source models, such as LLAMA-3.1-70B, we access them through APIs provided by third-party websites [1].

4.2 Main Experiments

We present the experimental results for Task 1 and Task 2 in Table 3.

[1] https://cloud.siliconflow.cn/models

Base Model	Task 1 Locating Function				Task 2 Answering Change			
	Easy(Py.)	Hard(Py.)	Java	Avg.	Easy(Py.)	Hard(Py.)	Java	Avg.
GPT-TURBO-3.5	**85.10**	**32.98**	80.89	**66.32**	26.01	13.09	63.28	34.13
GPT-4O-MINI	77.21	21.99	84.77	61.32	18.73	6.28	68.95	31.32
GPT-4O	70.71	25.65	81.19	59.18	22.22	13.61	**75.22**	37.02
LLAMA-3.1-8B	70.71	21.99	67.16	53.29	16.16	2.09	53.13	23.79
LLAMA-3.1-70B	75.51	29.84	81.19	62.18	22.73	8.38	**75.22**	35.44
QWEN2.5-CODER-7B	66.16	15.71	79.40	53.76	16.92	1.05	56.42	24.8
DEEPSEEK-CHAT	78.48	26.17	**82.08**	62.24	**38.99**	**16.75**	70.44	**42.06**

Table 3: Experiment results for function locating and function change answering task. We **bold** the best result and underline the second-best result.

Overall Performance of Function Locating. The average locating success rate for the seven LLMs across two languages is 59.76%. Both Easy(Py.) and Java datasets show relatively high scores, with averages of 74.84% and 79.53%, respectively. However, for the Hard(Py.) dataset, where there are multiple incompatible functions, all models perform poorly. For example, QWEN2.5-CODER-7B only achieves a pass rate of 15.71%. We find that the low performance is due to the models' inability to successfully locate all incompatible functions, leading to missed cases.

The model with the best performance on the locating task is GPT-TURBO-3.5, with an average locating success rate of 66.32%. Compared with other models, it can better cope with the locating task of Python language, both Easy(Py.) and Hard(Py.) tasks achieve the first 85.10% and 32.98% pass rates.

Overall Performance of Answering Change. The pass rate for successfully identifying function changes is lower than that for locating, with an average score of 32.65% across the seven models. In our evaluation, we find that although the models may successfully locate the function, they often output incorrect changes. For example, when the change is a deprecating type, the model incorrectly classifies it as a replacement change, answering that the function has been replaced by another.

Although the success rate of most models in answering is only about half of their locating success rate, two models stand out: DEEPSEEK-CHAT and GPT-4O. The former achieves a response success rate of 42.06%, the highest among all models, while the latter reaches a response success rate of 37.02%, ranking second. DEEPSEEK-CHAT not only accurately identifies the type of change but also provides accurate responses regarding the version where the change occurs.

Overall Performance of Code Migration. Table 4 illustrates the results of code migration. In the code migration OLD2NEW scenario, the average pass@1 success rate for the seven LLMs at the easy difficulty level is 33.56%, while at the hard difficulty level, the pass@1 rate is 16.20%. As the number of attempts increases, the success rate rises significantly, with pass@5 reaching 45.5% for the easy difficulty and 26.47% for the hard difficulty.

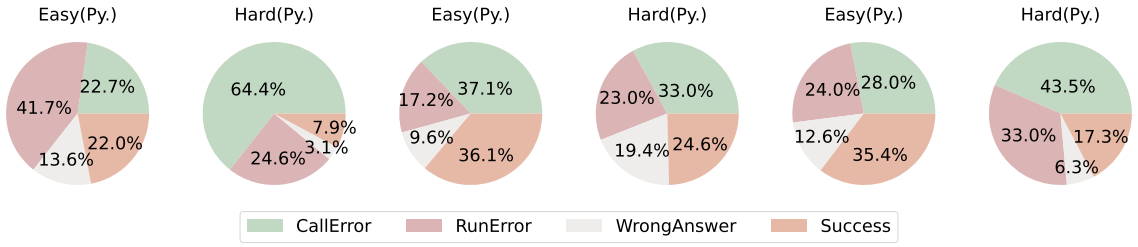
In the code migration NEW2OLD scenario, the average pass@1 and pass@5 success rate for the seven LLMs are only 12.77% and 17.30% at the hard difficulty level. In this case, increasing the number of model attempts did not improve the performance.

The model that performs best in the OLD2NEW migration is GPT-4O, achieving impressive results with pass@1 rate of 43.84% and pass@5 rate of 59.59% at the easy difficulty level. However, GPT-TURBO-3.5, which performs best in locating task, does not deliver outstanding results in code migration, especially at the hard difficulty level, where its pass@1 rate is only 7.32%. This shows that GPT-TURBO-3.5 is difficult to complete code migration, but it can help users find functions that are not compatible with the environment.

Preference of New Functions. We find that LLMs are more familiar with the new functions compared to the old ones. Our experimental results show that LLMs perform better in the OLD2NEW task compared to the NEW2OLD task. For example, GPT-4O achieves a pass@1 rate of 44.52% in the OLD2NEW task at easy difficulty, while for NEW2OLD at the same difficulty, it only reaches 28.00%. A possible reason for this is that the demand for writing code for new environments is more widespread, and during the train-

Base Model	Task 3 Migration (OLD2NEW)				Task 3 Migration (NEW2OLD)			
	Easy(Py.)		Hard(Py.)		Easy(Py.)		Hard(Py.)	
	Pass@1	Pass@5	Pass@1	Pass@5	Pass@1	Pass@5	Pass@1	Pass@5
GPT-TURBO-3.5	26.03	34.93	7.32	10.98	24.80	38.40	7.34	9.17
GPT-4O-MINI	30.82	49.32	15.85	26.83	29.60	**44.00**	11.93	16.51
GPT-4O	**43.84**	**59.59**	**26.83**	**47.56**	**31.60**	43.60	**22.94**	**27.52**
LLAMA-3.1-8B	23.97	28.08	8.54	10.97	20.80	24.00	7.34	11.93
LLAMA-3.1-70B	32.88	45.89	19.51	35.37	28.80	40.80	17.43	19.27
QWEN2.5-CODER-7B	32.19	46.58	14.63	24.39	29.20	38.00	8.26	12.84
DEEPSEEK-CHAT	41.20	54.11	20.73	29.27	29.60	39.60	14.68	23.85

Table 4: Experiment results for code migration, we report the results in two cases: OLD2NEW and NEW2OLD.



(a) Llama-3.1-8B-Instruct (b) GPT-4o (c) Deepseek-Chat

Figure 4: Error Analysis of Code Migration. `CallError` represents a function where an incompatible the environment is still called. `RunError` represents that the migrated code enters an infinite loop during execution. `WrongAnswer` represents this code runs normally and gets the result, but it is different from the standard answer. We combine the experimental results of NEW2OLD and OLD2NEW in this pie chart.

ing process, the proportion of new functions in the training data is higher than that of old functions, leading to this function preference. Furthermore, this trend varies in magnitude across different models. For instance, GPT-4O-MINI shows a smaller performance gap between NEW2OLD and OLD2NEW.

Error Analysis for Code Migration. As shown in Figure 4, a significant portion of the data failures can be attributed to `CallError`, where a function incompatible with the environment is still being invoked. For example, 50.8% of the code generated by LLAMA-3.1-8B for the Hard(Py.) migration task fails due to `CallError`. These errors occur either because the incompatible function was not successfully located or because, even when the model correctly identifies the incompatible function, it generates code that calls an incompatible one instead.

Another portion of the failed code is due to `RunError`, where the code compiles correctly but enters an infinite loop, causing it to run for an excessively long time. For instance, 33.0% of the code generated by DEEPSEEK-CHAT failed due to this error.

Additionally, some migrated code, while calling functions compatible with the environment and passing compilation successfully, produces results that deviate from the expected output, leading to a `WrongAnswer`. For instance, 19.4% of the code generated by GPT-4O failed due `WrongAnswer`.

Case Studies. Figure 5 illustrates the responses of four models performing the locating and answering change tasks. In this example, two models fail to locate, namely LLAMA-3.1-8B and GPT-TURBO-3.5. They locate another function `np.array2string` in the code. These two models provide the change knowledge for this function at version of 1.17 and 1.18, but the target environment is 1.16, which does not affect the function's operation in the target environment. Although this function did undergo a change in NumPy 1.18, it does not affect its functionality in the target version, Numpy 1.16. From this example, we can see that one possible reason for the model's error is an incorrect version comparison, specifically the confusion between the sizes of NumPy 1.16 and 1.18.

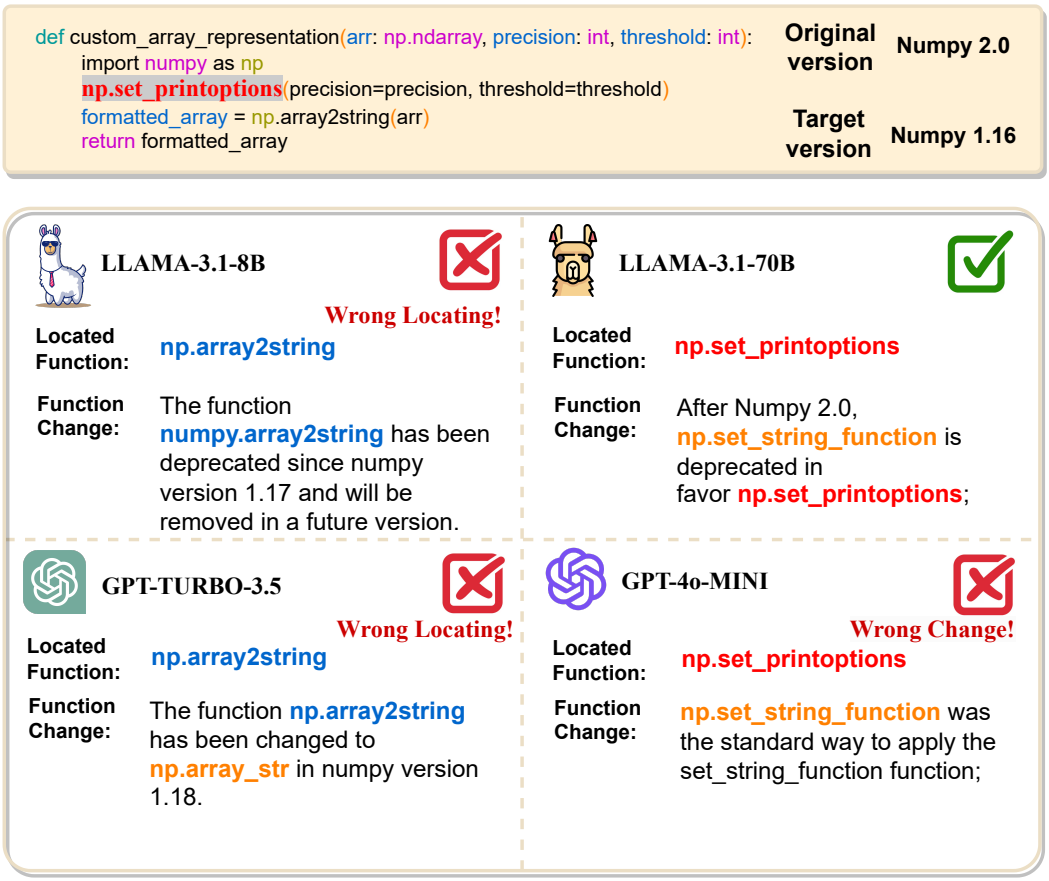


Figure 5: Case Study. We plot an example from Easy (Python) datasets and present the response of task-1 and task-2 for four LLMs. In this case study, we observe the phenomenon of logical inconsistency, where LLAMA-3.1-8B and GPT-TURBO-3.5 provide function changes that are unrelated to the migration process.

Two models, LLAMA-3.1-70B and GPT-4O-MINI, successfully locate functions `np.set_printoptions` that are incompatible with NumPy 1.16. However, GPT-4o-mini's response of function changes is unsatisfactory, as it failed to accurately provide the version in which the function change occurred. This problem is actually quite common in our evaluation, where the version number provided for the change is different from the actual version.

5 Conclusion

In this paper, we introduce CODEMENV, a novel benchmark designed to evaluate whether language models can perform code migration, i.e., adapting code to the desired environment. CODEMENV provides three tasks that systematically assess the model's ability to accurately locate incompatible functions, answering function change knowledge, and, finally, correctly migrate the code.

We systematically evaluate seven LLMs on CODEMENV, and experimental results show the model's understanding of old version functions is lower than that of new version functions, making it more difficult to perform the NEW2OLD migration task effectively. In addition, through detailed error analysis, we reveal the phenomenon of logical inconsistency in code migration, where the changes provided by the model are not helpful for our migration task.

In future, we plan to expand the dataset to include more packages and programming languages, as well as increase the code length to a repository level.

Limitations

CODEMENV is relatively small, particularly the Java dataset. Additionally, the language features of Java make it challenging to establish rigorous unit tests. CODEMENV currently involves only two programming languages, Python and Java. We plan to add more programming languages in the future.

Ethics Statement

Throughout our work, we have strictly adhered to ethical standards. The creation of our dataset

also complies with open-source regulations, and the data has undergone manual checks to prevent harmful content.

References

The claude 3 model family: Opus, sonnet, haiku.

Marah Abdin, Sam Ade Jacobs, Ammar Ahmad Awan, Jyoti Aneja, Ahmed Awadallah, Hany Hassan Awadalla, Nguyen Bach, Amit Bahree, Arash Bakhtiari, Harkirat Singh Behl, Alon Benhaim, Misha Bilenko, and Johan Bjorck. 2024. Phi-3 technical report: A highly capable language model locally on your phone. *ArXiv*, abs/2404.14219.

Abhinav Jauhri Abhimanyu Dubey et al. 2024. The llama 3 herd of models. *ArXiv*, abs/2407.21783.

Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*.

Keyuan Cheng, Gang Lin, Haoyang Fei, Yuxuan Zhai, Lu Yu, Muhammad Asif Ali, Lijie Hu, and Di Wang. 2024. Multi-hop question answering under temporal knowledge editing. *ArXiv*, abs/2404.00492.

Xueying Du, Mingwei Liu, Kaixin Wang, Hanlin Wang, Junwei Liu, Yixuan Chen, Jiayi Feng, Chaofeng Sha, Xin Peng, and Yiling Lou. 2024. Evaluating large language models in class-level code generation. *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE)*, pages 982–994.

Hasan Ferit Eniser, Hanliang Zhang, Cristina David, Meng Wang, Maria Christakis, Brandon Paulsen, Joey Dodds, and Daniel Kroening. 2024. Towards translating real-world code with llms: A study of translating to rust. *ArXiv*, abs/2405.11514.

Kelvin Guu, Kenton Lee, Zora Tung, Panupong Pasupat, and Ming-Wei Chang. 2020. Realm: Retrieval-augmented language model pre-training. *ArXiv*, abs/2002.08909.

Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Xiaodong Song, and Jacob Steinhardt. 2021. Measuring coding challenge competence with apps. *ArXiv*, abs/2105.09938.

Cheng-Yu Hsieh, Sibei Chen, Chun-Liang Li, Yasuhisa Fujii, Alexander J. Ratner, Chen-Yu Lee, Ranjay Krishna, and Tomas Pfister. 2023. Tool documentation enables zero-shot tool-usage with large language models. *ArXiv*, abs/2308.00675.

Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. 2024. A survey on large language models for code generation. *ArXiv*, abs/2406.00515.

Patrick Lewis, Ethan Perez, Aleksandara Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Kuttler, Mike Lewis, Wen tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. 2020. Retrieval-augmented generation for knowledge-intensive nlp tasks. *ArXiv*, abs/2005.11401.

Xiaopeng Li, Shangwen Wang, Shasha Li, Jun Ma, Jie Yu, Xiaodong Liu, Jing Wang, Bing Ji, and Weimin Zhang. 2024. Model editing for llms4code: How far are we? *ArXiv*, abs/2411.06638.

Zeyu Leo Liu, Shrey Pandit, Xi Ye, Eunsol Choi, and Greg Durrett. 2024. Codeupdatearena: Benchmarking knowledge editing on api updates. *ArXiv*, abs/2407.06249.

Kevin Meng, David Bau, Alex Andonian, and Yonatan Belinkov. 2022a. Locating and editing factual associations in gpt. *Advances in Neural Information Processing Systems*, 35:17359–17372.

Kevin Meng, Arnab Sen Sharma, Alex J Andonian, Yonatan Belinkov, and David Bau. 2022b. Mass-editing memory in a transformer. In *The Eleventh International Conference on Learning Representations*.

OpenAI, Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, et al. 2024a. Gpt-4 technical report.

OpenAI, Aaron Hurst, Adam Lerer, Adam P. Goucher, Adam Perelman, Aditya Ramesh, Aidan Clark, AJ Ostrow, Akila Welihinda, Alan Hayes, Alec Radford, Aleksander Mądry, Alex Baker-Whitcomb, Alex Beutel, et al. 2024b. Gpt-4o system card.

Machel Reid, Nikolay Savinov, Denis Teplyashin, Dmitry Lepikhin, Timothy P. Lillicrap, Jean-Baptiste Alayrac, Radu Soricut, Angeliki Lazaridou, Orhan Firat, Julian Schrittwieser, Ioannis Antonoglou, Rohan Anil, Sebastian Borgeaud, and Andrew M. 2024. Gemini 1.5: Unlocking multimodal understanding across millions of tokens of context. *ArXiv*, abs/2403.05530.

Zhihong Shao, Damai Dai, Daya Guo, Bo Liu (Benjamin Liu), Zihan Wang, and Huajian Xin. 2024. Deepseek-v2: A strong, economical, and efficient mixture-of-experts language model. *ArXiv*, abs/2405.04434.

Hongjin Su, Shuyang Jiang, Yuhang Lai, Haoyuan Wu, Boao Shi, Che Liu, Qian Liu, and Tao Yu. 2024. Evor: Evolving retrieval for code generation. In *Conference on Empirical Methods in Natural Language Processing*.

Qwen Team. 2024. Qwen2.5: A party of foundation models.

Junjie Ye, Xuanting Chen, Nuo Xu, Can Zu, Zekai Shao, Shichun Liu, Yuhan Cui, Zeyang Zhou, Chao Gong, Yang Shen, Jie Zhou, Siming Chen, Tao Gui,

Qi Zhang, and Xuanjing Huang. 2023. A comprehensive capability analysis of gpt-3 and gpt-3.5 series models.

Zhiqiang Yuan, Weitong Chen, Hanlin Wang, Kai Yu, Xin Peng, and Yiling Lou. 2024. Transagent: An llm-based multi-agent system for code translation. *ArXiv*, abs/2409.19894.

Zexuan Zhong, Zhengxuan Wu, Christopher D Manning, Christopher Potts, and Danqi Chen. 2023. Mquake: Assessing knowledge editing in language models via multi-hop questions. *arXiv preprint arXiv:2305.14795*.

Shuyan Zhou, Uri Alon, Frank F. Xu, Zhiruo Wang, Zhengbao Jiang, and Graham Neubig. 2022. Docprompting: Generating code by retrieving the docs. In *International Conference on Learning Representations*.

A Prompts for CODEMENV

See Prompt1 for the generation of python code.

See Prompt2 for the evaluation of python of various llms.

See Prompt3 for the judgment of Python code evaluation results for llms.

See Prompt4 for the generation of unit test.

See Prompt5 for the improving of unit test.

See Prompt6 for the generation of java code.

See Prompt7 for the evaluation of java of various llms.

See Prompt8 for the judgment of Java code evaluation results for llms.

Prompt 1: Generate Python Code based on Given function

==== SYSTEM ====
You are a very experienced programmer who is familiar with the functions of many functions and is good at applying them. At the same time, you are thoughtful and creative and like to apply some functions to solve algorithmic problems.

First of all, I will give you an existing library function, you will get the function with signatures and functionality, as well as import methods. I hope you can think about the application of this library function according to the function of this library function, be bold and creative, and then write a piece of code that calls this new function, we call this code the solution. This solution is a function. This solution should be able to solve medium and difficult algorithmic problems, which require "multiple inferences", at least three or four steps to solve, rather than simply calling your library function. There should be no comments in this solution.

Then, design a problem for the solution you generated, and ask others to be able to generate a solution from the problem , and your problem description should be biased towards the functionality of the solution, as well as the inputs and outputs, rather than leading to a step-by-step solution generation in the description. You should be clear about the data type and dimension of each parameter you input, as well as the data type and parameters of the output. Your problem description goes like "Please use python code to help me with a function...". Indicate in the description which library is being called, but not which library function is being called. Don't mention the details of the task fulfillment.

Note: Do not alias when importing; Here's a return template,only output the JSON in raw text. Don't return anything else.

{
solution_function: The function that you generate. Make sure the code you return is runnable.
solution_signature: The signature of the function you generated, indicating the input and output. And the name of the solution should derive from its functionality,
problem: Generate a literal description of this function. Describe the data type and dimensions of each input parameter and the data type and dimension of the output.
}

==== USER ====
The package name for the new library function is:
<PACKAGE>
The import method is as follows:
<IMPORT>
The signature of the new library function is:
<SIGNATURE>
The feature description of the new library function is:
[DOC]
<DOC_STRING>
[/DOC]
Note: Do not alias when importing; Only output the JSON in raw text.Don't return anything else.

Prompt 2: Evaluate the ability of large language models to locate and correct errors in the python code.

==== SYSTEM ====
You're a good assistant, and now you need to help me change the code.
I'll give you a piece of python code that has an error due to an exception in the python library version. You need to locate the wrong API in this code. Then give the information about the change of the wrong api. It must include the change type deprecation/addition/replacement,the replace api(if the change type is replace), and the version of the API that changed. Finally return your corrected code, noting that you only need to fix the wrong API in the code, but your return includes the entire code as modified.
Note that Only output the JSON in raw text. Don't return anything else. And here's an example of what you returned.
{
ai_api_wrong: There is the wrong API in the code because of the version,
ai_api_change: 1.The specified API(error api) has changed due to version changes, such as being added in version..., being abandoned in version..., or the calling method has changed; 2.The replace method is... 3.The version tha the api changed is...
code_fixed: Entire code modified
}

Here's an example of an answer.
{
ai_api_wrong: numpy.compare_chararrays.
ai_api_change: 1.replacement 2.use numpy.char.compare_chararrays instead
3.The function numpy.compare_chararrays has been removed in numpy version 2.0.
code_fixed: def string_array_similarities(strings1, strings2):
result = []
for s1 in strings1:
temp_result = 0
for s2 in strings2:
length_diff = abs(len(s1) - len(s2))
comparison = numpy.char.compare_chararrays(numpy.array(list(s1)), numpy.array(list(s2)), cmp='==', assume_equal=False)
similarity = numpy.sum(comparison) - length_diff
temp_result = max(temp_result, similarity)
result.append(temp_result)
return result
}

==== USER ====
Here's the code you need to identify errors.
[CODE]
<CODE>
[/CODE]
Here's the Python library you need to modify your code.
[PACKAGE]
<PACKAGE>
[/PACKAGE]
Here's the version of above package.
[VERSION]
<VERSION>
[/VERSION]

Prompt 3: Judge the correctness of AI's reasoning of python code.

==== SYSTEM ====
You are a good helper for a human being. I'm now having another big AI model try to figure out the API in a piece of code that is incorrectly called because of the version, and have it return the error api,the api change about different version and code after he fixed. I'll give you another reason for the AI's return, and I'll give you a basis for judging whether the AI's reason is correct. I'm going to give you a judgment based on the python library function that is wrong and whether the version of the library function is too low or too high.

Please compare the wrong apis returned by the AI and the correct apis I give you. If ai_api_wrong contains api_wrong, the judge_locate_answer is 1,unless return 0.
Compare whether the change of the api returned by the and the real change I give you. You can loosely compare the two changes. If they are related or only have a little difference, the judge_update_answer is 1. If two changes are absolutly are completely irrelevant, return 0. Remenber if judge_locate_answer is 0, judge_update_answer must be 0. Note that Only output the JSON in raw text. Don't return anything else. And here's an example of what you returned.
{
judge_reason: The reason why the AI determines whether it is correct or wrong,
judge_locate_answer: {0/1}
judge_update_answer: {0/1}
}

==== USER ====
Here's the code that lets the AI judge that there is an error.
[CODE]
<CODE>
[/CODE]
Here are the apis given by LLM that are not suitable for the target environment.
[API_LOCATE_BY_LLM]
<API_LOCATE_BY_LLM>
[API_LOCATE_BY_LLM]
Here's the information regarding the changes in this API, which was returned by LLM.
[CHANGE_INFORMATION_BY_LLM]
<CHANGE_INFORMATION_BY_LLM>
[CHANGE_INFORMATION_BY_LLM]
Here are the answers.
[API_REFERENCE_ANSWER]
<API_REFERENCE_ANSWER>
[API_REFERENCE_ANSWER]
[CHANGE_INFORMATION_REFERENCE_ANSWER]
<CHANGE_INFORMATION_REFERENCE_ANSWER>
[CHANGE_INFORMATION_REFERENCE_ANSWER]
The version is too high or too low.
[VERSION_ERROR]
<VERSION_ERROR>
[/VERSION_ERROR]

Prompt 4: Generate test data

==== SYSTEM ====
Role
A very experienced programmer who is good at algorithmic reasoning and can write high-quality code.

Responsibilities
Write 3 sets of *high-quality* and *comprehensive* input test data based on the problem description and benchmark code.

The specific description of these requirements is as follows:

Problem:
That is, the problem situation. The type of input data and the range limit of the input data are often given in the problem.
(Problem is between "[PROBLEM]" and "[/PROBLEM]")

Benchmark code:
That is, the given callable code, and its parameters are each set of input data to be passed in (Benchmark code is between "[CODE]" and "[/CODE]")

Implementation steps
Please answer the questions strictly according to the above requirements and the following steps:

1. Determine the input data
- First analyze the problem and the given code to determine the type of input data,

2. Final input data group generation
Based on step 1, return the string of the input data group
- Return format: case1:

====== Task start =====
Below is the given problem and function.

==== USER ====
[PROBLEM]
<PROBLEM>
[/PROBLEM]
[CODE]
<CODE>
[/CODE]

Prompt 5: Improve data quality

==== SYSTEM ====
Role
An experienced data tester who is good at writing more accurate and higher quality test data based on error information.

Responsibilities
Adjust the test data group according to the provided executable script and running information, and return the adjusted test data

Executable script:
That is, a script that can be compiled and run, and the script code already contains an array of test data.(BETWEEN "[TARGET_IMPLEMENTATION]" and "[/TARGET_IMPLEMENTATION]")

Running information: That is, the running information of each set of test data when the function is running, mainly focusing on error information.(BETWEEN "[MESSAGE]" and "[/MESSAGE]")

Example:
- input:
[TARGET_IMPLEMENTATION]

```
import tensorflow as tf

def weighted_average_division(matrix, weights):
    matrix_tensor = tf.convert_to_tensor(matrix, dtype=tf.float32)
    weights_tensor = tf.convert_to_tensor(weights, dtype=tf.float32)
    weighted_matrix = tf.multiply(matrix_tensor, weights_tensor)
    sum_of_weights = tf.reduce_sum(weights_tensor)
    weighted_sum = tf.reduce_sum(weighted_matrix)
    return tf.divide(weighted_sum, sum_of_weights).numpy()

# Input data
test_data = [
        ([[1.0, 2.0], [3.0, 4.0]], [0.5, 0.5]),
        ([[1.5, 2.5], [3.5, 4.5], [5.5, 6.5]], [0.2, 0.3, 0.5]),
        ([[10.0]], [1.0])
    ]

# Process each data case and save results
results = []
for matrix, weights in test_data:
    try:
        result = weighted_average_division(matrix, weights)
        results.append(result)
    except Exception as e:
        results.append(f"error:{e}")
```

[/TARGET_IMPLEMENTATION]
[MESSAGE]
"""
5.0
error:function_node __wrapped__Mul_device_/job:localhost/replica:0/task:0/device:CPU:0 Incompatible shapes: [3,2] vs. [3] [Op:Mul]
10.0
"""
[/MESSAGE]

- output:
case1:[[1.0, 2.0], [3.0, 4.0]], [0.5, 0.5],
case2:[[-1.0, -2.0], [-3.0, -4.0]], [0.5, 0.5],
case3:[[10.0]], [1.0]

Notes
Here, you only need to pay attention to the test data with running errors. For arrays without error information records, there is no need to adjust.

Implementation steps
Please strictly follow the above requirements and the following steps to answer the questions:
1. Test data extraction and identification
-Extract the parameters passed by the calling function from the executable script as the test data group

Prompt 5: Improve data quality

2. Match the test data group with the corresponding operation information
-Pair the test data input groups in sequence according to the operation results
3. Save the test data group that runs correctly and replace the test data group that runs incorrectly
-Keep the test data group that runs correctly unchanged
-For the test data group that runs incorrectly, analyze the cause according to the error information, avoid similar errors, and replace the new test data group
4. Finally, just return the modified test data, do not return unnecessary explanations!

====== Task start =====
Below is the given executable script and running information.

==== USER ====
[TARGET_IMPLEMENTATION]
<TARGET_IMPLEMENTATION>
[/TARGET_IMPLEMENTATION]
[MESSAGE]
<MESSAGE>
[/MESSAGE]

Prompt 6: Generate Java Code based on Given function

==== SYSTEM ====
You are a very experienced JAVA programmer who is familiar with various library functions of java and is good at applying them. At the same time, you are thoughtful and creative, and like to apply some functions to solve algorithmic problems.

First of all, I will specify that you use an old API to complete a class, this API may have been removed in the new JDK. Assuming that I am running in an old JDK environment, please call the API anyway.

Then, generate a functional description for your generated code, and I can ask others to be able to generate the code from the problem.

Note: Do not alias when importing; Here's a return template,only output the JSON in raw text. Don't return anything else.

{
java_code: The function that you generate. Make sure the code you return is runnable.
class_name: The name of the class you generate.
function_description: The function description of your generated code.
}

==== USER ====
The sigature of the new library function is: **<SIGNATURE>**

Note: Do not alias when importing; Only output the JSON in raw text.Don't return anything else.

Prompt 7: Evaluate the ability of large language models to locate and correct errors in the java code.

==== SYSTEM ====
You're a good assistant, and now you need to help me find the error of the code. I'll give you a piece of java code that has errors due to an exception in the java JDK version. You need to locate the wrong APIs in this code, and explain what version changes have taken place in the API that caused the error you pointed out.
Note that your answers must be concise, and you only need to point out the mistake directly.
Here's an example of an answer:
Output:
ai_api_wrong: com.sun.javadoc.AnnotatedType
ai_api_change: The declarations in this package have been superseded by those in the package jdk.javadoc.doclet. For more information, see the Migration Guide in the documentation for that package.
==== USER ====
Here's the code you need to identify errors.
[CODE]
<CODE>
[/CODE]
Here's the version of the JDK
[VERSION]
<VERSION>
[VERSION]

Prompt 8: Judge the correctness of AI's reasoning of java code.

==== SYSTEM ====
You are a good helper for a human being. I'm now having another large AI model try to figure out the API in a piece of code that is incorrectly called because of the JDK version, and have it return the error_api,the api's change about different version. I'm going to give you a judgment based on the java api(class or interface) that is deparcted.

Please compare the wrong apis returned by the AI and the correct apis I give you. If api_locate_by_llm contains api_reference_answer, the judge_locate_answer is 1,unless return 0.
Compare whether the change of the api returned by the and the real change I give you. You can loosely compare the two changes. If they are related or only have a little difference, the judge_update_answer is 1. If two changes are absolutly are completely irrelevant, return 0. Remenber if judge_locate_answer is 0, judge_update_answer must be 0.
{
judge_reason: The reason why the AI determines whether it is correct or wrong,
judge_locate_answer: {0/1}
judge_update_answer: {0/1}
}

==== USER ====
Here's the code that lets the AI judge that there is an error.
[CODE]
<CODE>
[/CODE]
Here's the wrong apis that the AI returned.
[API_LOCATE_BY_LLM]
<API_LOCATE_BY_LLM>
[API_LOCATE_BY_LLM]
Here's the change of the wrong apis that the AI returned.
[CHANGE_INFORMATION_BY_LLM]
<CHANGE_INFORMATION_BY_LLM>
[CHANGE_INFORMATION_BY_LLM]
Here are the answers.
[API_REFERENCE_ANSWER]
<API_REFERENCE_ANSWER>
[API_REFERENCE_ANSWER]
[CHANGE_INFORMATION_REFERENCE_ANSWER]
<CHANGE_INFORMATION_REFERENCE_ANSWER>
[CHANGE_INFORMATION_REFERENCE_ANSWER]
The version is too high or too low.
[VERSION_ERROR]
<VERSION_ERROR>
[/VERSION_ERROR]

B Datsets Statistics

Datasets	jdk.nashorn	org.xml	com.sun	java.applet	java.beans	java.rmi	java.util	java.security
Java	188	9	86	9	3	15	7	18

Table 5: Distribution of Code Segments across Various Java Packages

Datasets	numpy	python	math	re	os	random	itertools	torch	tensorflow	pandas	csv
Easy(Py.)	39	26	51	5	34	3	15	21	154	46	2
Hard(Py.)	20	-	-	-	-	-	-	21	115	35	-

Table 6: Distribution of Code Segments across Various Python Packages

C Data Example

C.1 Data Example of Easy(Py.)

Related API

```
np.row_stack(tup, *, dtype=None, casting='same_kind')->numpy.ndarray
```

Update Message
np.row_stack has been deprecated to reduce redundancy and encourage direct usage of np.vstack.

Document Message
np.row_stack was used as an alias for np.vstack, which vertically stacks arrays row-wise.

Original And Target Environment

```
Original version: numpy1.26
Target version: numpy2.0
```

Code Before Migration

```
def maximize_channel_sum(matrices):
    import numpy as np
    stacked_matrix = np.row_stack(matrices)
    return np.max(np.sum(stacked_matrix, axis=0))
```

Problem Description
Please use python code to help me with a function that takes a list of 2D numpy arrays, all having the same number of columns, and returns the maximum sum of any column after stacking all arrays vertically.
Each array in the list is a 2D numpy array, and the output is a single integer representing the maximum column sum.
Use the numpy library for the operations.

Incompatible Function

```
numpy.row_stack
```

Function Change
The function numpy.row_stack has been removed in numpy version 2.0 instead stack function is recommended.

Code After Migration

```
def maximize_channel_sum(matrices):
    import numpy as np
    stacked_matrix = np.stack(matrices, axis=0).reshape(-1,
    ↪  matrices[0].shape[1])
    return np.max(np.sum(stacked_matrix, axis=0))
```

C.1 Data Example of Easy(Py.)

Unit Test

```
test_data = [
    [np.array([[1, 2], [3, 4]]), np.array([[5, 6], [7, 8]])],
    [np.array([[10, 20, 30], [40, 50, 60]]), np.array([[5, 15, 25], [35, 45,
    ↪  55]]), np.array([[1, 2, 3], [4, 5, 6]])],
    [np.array([[10, -5], [-2, 3]]), np.array([[5, -10], [8, 10]])]
]
```

Complete Code

```
import numpy as np

def maximize_channel_sum(matrices):
    import numpy as np
    stacked_matrix = np.stack(matrices, axis=0).reshape(-1,
    ↪  matrices[0].shape[1])
    return np.max(np.sum(stacked_matrix, axis=0))

# Input data
test_data = [
    [np.array([[1, 2], [3, 4]]), np.array([[5, 6], [7, 8]])],
    [np.array([[10, 20, 30], [40, 50, 60]]), np.array([[5, 15, 25], [35, 45,
    ↪  55]]), np.array([[1, 2, 3], [4, 5, 6]])],
    [np.array([[10, -5], [-2, 3]]), np.array([[5, -10], [8, 10]])]
]

for matrices in test_data:
    try:
        result = maximize_channel_sum(matrices)
        print(result)
    except Exception as e:
        print("error:", e)
```

Operation Results

```
20
179
21
```

C.2 Data Example of Hard(Py.)

Related API

```
pd.bfill()
pd.api.types.is_any_real_numeric_dtype(arr_or_dtype)->bool
```

Update Message
1)Before pandas 2.0, pd.Series.backfill was the standard way to apply the backfill function; however, after pandas 2.0, it is recommended to use pd.bfill instead.
2)New in pandas 2.0.

Document Message
1)It is used to backward-fill missing values in a Series.
2)Check whether the provided array or dtype is of a real number dtype.

Original And Target Environment

```
Original version: pandas2.0
Target version: pandas1.0.0
```

Code Before Migration

```
def interpolate_and_check_numeric(data):
    df = pd.DataFrame(data)
    df = df.bfill()
    numeric_cols = [col for col in df.columns if
    ↪  pd.api.types.is_any_real_numeric_dtype(df[col])]
    numeric_data = df[numeric_cols]
    return numeric_data.mean().to_dict()
```

Problem Description
Please use python code to help me with a function that takes a dictionary representing a dataset with possible missing values in its columns. Each key in the dictionary is a column name, and the value is a list representing column data. The function should fill in missing values by carrying backward the next valid observation. Then, identify which columns contain numeric data and return a dictionary with the mean of each numeric column after filling missing values. Use the pandas library in your solution. The input is a dictionary where keys are strings and values are lists of equal length, which may contain None to represent missing data. The output is a dictionary where keys are column names containing numeric data and values are their respective means as floats.

Incompatible Function

```
pd.bfill
pd.api.types.is_any_real_numeric_dtype
```

Function Change
In Pandas version 1.0.0, DataFrame.bfill is not directly callable via pd.bfill since it's a method of DataFrame objects.
pd.api.types.is_any_real_numeric_dtype was introduced in later versions;
in version 1.0.0, use pd.api.types.is_numeric_dtype instead.

Code After Migration

```
import pandas as pd

def interpolate_and_check_numeric(data):
    df = pd.DataFrame(data)
    df = df.bfill()
    numeric_cols = [col for col in df.columns if
    ↪  pd.api.types.is_numeric_dtype(df[col])]
    numeric_data = df[numeric_cols]
    return numeric_data.mean().to_dict()
```

C.2 Data Example of Hard(Py.)

Unit Test

```
test_data = [
    {
        'A': [1.2, None, 3.4, None, 5.6],
        'B': [None, 10, None, 12, None],
        'C': ['a', 'b', 'c', 'd', 'e'],
        'D': [None, None, 2.2, None, 3.3]
    },
    {
        'Height': [170, 165, 180, 175],
        'Weight': [70, 60, 80, 75],
        'Labels': ['tall', 'short', 'tall', 'medium']
    },
    {
        'A': [None, None, None],
        'B': [None, None, None],
        'C': [None, None, None]
    }
]
```

Complete Code

```
import pandas as pd

def interpolate_and_check_numeric(data):
    df = pd.DataFrame(data)
    df = df.bfill()
    numeric_cols = [col for col in df.columns if
    ↪  pd.api.types.is_any_real_numeric_dtype(df[col])]
    numeric_data = df[numeric_cols]
    return numeric_data.mean().to_dict()

# Input data
test_data = [
    {
        'A': [1.2, None, 3.4, None, 5.6],
        'B': [None, 10, None, 12, None],
        'C': ['a', 'b', 'c', 'd', 'e'],
        'D': [None, None, 2.2, None, 3.3]
    },
    {
        'Height': [170, 165, 180, 175],
        'Weight': [70, 60, 80, 75],
        'Labels': ['tall', 'short', 'tall', 'medium']
    },
    {
        'A': [None, None, None],
        'B': [None, None, None],
        'C': [None, None, None]
    }
]

for data in test_data:
    try:
        result = interpolate_and_check_numeric(data)
        print(result)
    except Exception as e:
        print("error:", e)
```

Operation Results

```
{'A': 3.84, 'B': 11.0, 'D': 2.6399999999999997}
{'Height': 172.5, 'Weight': 71.25}
{}
```

C.3 Data Example of Java

Related API

```
jdk.nashorn.api.tree.WhileLoopTree
```

Update Message
Nashorn JavaScript script engine and APIs, and the jjs tool are deprecated with the intent to remove them in a future release.

Original And Target Environment

```
Original version: jdk1.7
Target version: jdk11
```

Code

```
import jdk.nashorn.api.tree.WhileLoopTree;
import jdk.nashorn.api.tree.Tree;
import jdk.nashorn.api.tree.ExpressionTree;
import jdk.nashorn.api.tree.StatementTree;
public class OldWhileLoopTreeExample {
  public void analyzeWhileLoop(Tree tree) {
  if (tree instanceof WhileLoopTree) {
  WhileLoopTree whileLoop =(WhileLoopTree) tree;
        ExpressionTree condition = whileLoop.getCondition();
        StatementTree statement = whileLoop.getStatement();
        System.out.println("While Loop Condition: " + condition.toString());
        System.out.println("While Loop Statement: " +
        statement.toString());        }
        else {
        System.out.println("The provided tree is not a WhileLoopTree.");
           }
        }
    public static void main(String[] args) {
        Tree someTree = null;

        OldWhileLoopTreeExample example = new OldWhileLoopTreeExample();
        example.analyzeWhileLoop(someTree);
    }
    }
```

Function Description
This class, OldWhileLoopTreeExample, demonstrates the usage of the deprecated Nashorn API's WhileLoopTree class. It includes a method analyzeWhileLoop that takes a Tree object as input, checks if it is an instance of WhileLoopTree, and if so, extracts and prints the condition and statement of the while loop. The main function provides a framework for how this method might be used, although actual tree creation is complex and not included in this example.

Incompatible Function

```
jdk.nashorn.api.tree.WhileLoopTree
```

Function Change
The Nashorn JavaScript engine and the 'jdk.nashorn.api' package were deprecated in JDK 11 and removed in JDK 17. For more information, see the Java API documentation for Nashorn's removal.

D Further Experiments

Performance across Different Packages. As Figure 6 show, LLMs show significant performance differences across different packages. For example, GPT-4o achieves a pass@1 rate of around 71% in Pandas package, but only 0% and 20% pass@1 rates in CSV and TensorFlow package, respectively. One possi-

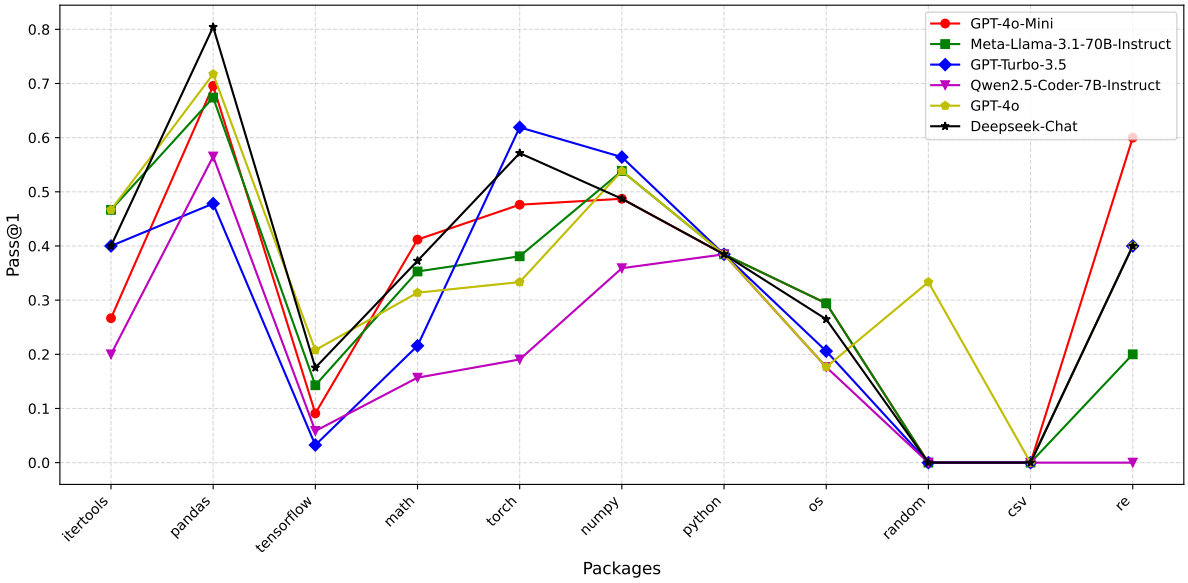


Figure 6: The experimental results of code migration at easy difficulty across different packages.

ble reason for this discrepancy is the varying difficulty of data across different packages. Additionally, the extent to which different packages are covered in the LLM's training data may also contribute to the performance differences.