CodeMEnv: Benchmarking Large Language Models on Code Migration

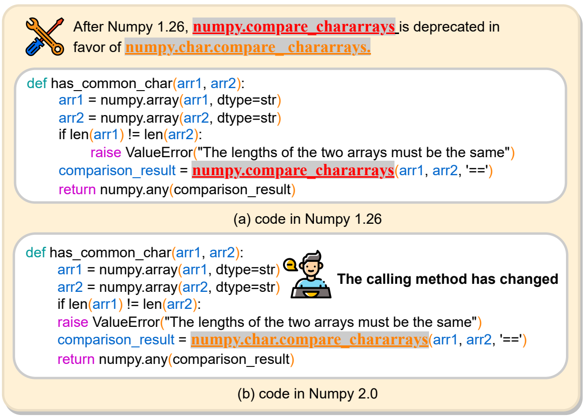
Keyuan Cheng[*,1,3,4], **Xudong Shen**[*,1,4], **Yihao Yang**[*,1,4], **Tengyue Wang**[1,4], **Yang Cao**[1,4], **Muhammad Asif Ali**[2], **Hanbin Wang**[3], **Lijie Hu**[†,1,2], **Di Wang**[†,1,2]
[1]Provable Responsible AI and Data Analytics (PRADA) Lab
[2]King Abdullah University of Science and Technology
[3]Peking University [4]South China University of Technology

Abstract

Large language models (LLMs) have shown remarkable capabilities across various software engineering tasks; however, their effectiveness in code migration—adapting code to run in different environments—remains insufficiently studied. In this work, we introduce CodeMEnv: **Code** **M**igration Across **Env**ironment, a new benchmark specifically designed to assess LLMs' abilities in code migration scenarios. CodeMEnv consists of 922 examples spanning 19 Python and Java packages, and covers three core tasks: (1) identifying functions incompatible with specific versions, (2) detecting changes in function definitions, and (3) adapting code to target environments. Experimental evaluation with seven LLMs on CodeMEnv yield an average pass@1 rate of 26.50%, with GPT-4o achieving the highest score at 43.84%. Key findings include: (i) LLMs tend to be more proficient with newer function versions, which aids in migrating legacy code, and (ii) LLMs sometimes exhibit logical inconsistencies by identifying function changes irrelevant to the intended migration environment. The datasets are available at `https://github.com/xdshen-ai/Benchmark-of-Code-Migration`.

1 Introduction

Large Language Models (LLMs) have demonstrated remarkable abilities in software engineering tasks, including code generation (Jiang et al., 2024; Du et al., 2024) and code translation (Yuan et al., 2024; Eniser et al., 2024). General-purpose LLMs such as GPT-4 (Achiam et al., 2023), Claude-3 (The), and DeepSeek (Shao et al., 2024) consistently achieve state-of-the-art results across diverse programming challenges, often outperforming traditional approaches on es-


After Numpy 1.26, numpy.compare_chararrays is deprecated in favor of numpy.char.compare_chararrays.

```
def has_common_char(arr1, arr2):
    arr1 = numpy.array(arr1, dtype=str)
    arr2 = numpy.array(arr2, dtype=str)
    if len(arr1) != len(arr2):
        raise ValueError("The lengths of the two arrays must be the same")
    comparison_result = numpy.compare_chararrays(arr1, arr2, '==')
    return numpy.any(comparison_result)
```

(a) code in Numpy 1.26

```
def has_common_char(arr1, arr2):
    arr1 = numpy.array(arr1, dtype=str)
    arr2 = numpy.array(arr2, dtype=str)
    if len(arr1) != len(arr2):
    raise ValueError("The lengths of the two arrays must be the same")
    comparison_result = numpy.char.compare_chararrays(arr1, arr2, '==')
    return numpy.any(comparison_result)
```

(b) code in Numpy 2.0

Figure 1: The function `compare_chararrays` underwent changes after Numpy 1.26, creating compatibility issues between NumPy 1.26 and 2.0.

tablished benchmarks. Beyond these general models, a range of specialized CodeLLMs have been introduced to further advance performance on code-related tasks. Notable examples include CodeT5+ (Wang et al., 2023), CodeLlama (Rozière et al., 2023), and StarCoder2 (Lozhkov et al., 2024). Thanks to their tailored architectures and training data, these models exhibit a deep understanding of code structure and syntax, frequently surpassing general LLMs on programming-specific benchmarks.

Despite the impressive progress of LLMs in various code-related tasks, their ability to perform code migration—adapting code to run correctly in a different environment—remains largely unexplored. Code migration is a critical challenge in practical software development. For example, when users attempt to run code obtained from sources like GitHub, they often encounter compatibility issues that require significant manual effort to resolve. If LLMs could automate the migration of code to fit a user's existing environment, it would greatly streamline the process of environment setup and reduce the burden of manual configuration.

The root cause of these compatibility issues

[*]Equal Contribution.
[†]Corresponding Author.

lies in the ongoing evolution and versioning of software libraries. As libraries are updated and maintained, APIs and function calls may change, leading to incompatibilities across different environments. For instance, as illustrated in Figure 1, the function call `compare_chararrays` was moved from the `numpy` package to the `numpy.char` submodule. This change results in functionally equivalent code being implemented differently in NumPy 1.26 versus NumPy 2.0, highlighting the challenges of code migration across library versions.

Research on code migration is still in its infancy, with the majority of prior work concentrating on cross-language migration (i.e., code translation) rather than migration across different library versions or environments. Only a few recent efforts, such as that by Google researchers (Ziftci et al., 2025), have explored automated LLM-based approaches for identifying code changes required for cross-version migration. However, there remains a significant gap: the absence of comprehensive benchmarks to systematically assess the code migration capabilities of LLMs.

To address this gap, we introduce a new benchmark, CODEMENV (**Code** **M**igration Across **Env**ironment). CODEMENV is built from 922 real-world function changes, manually curated from official sources, spanning 19 Python and Java packages. As shown in Figure 2, the benchmark is organized into three tasks that collectively evaluate LLMs' abilities to perform code migration:

Task 1: Locate version-incompatible functions. Given a code snippet and a specified target environment, the model is tasked with identifying functions or code segments that may not be compatible with the running environment.

Task 2: Answering function changes. The model must explain the specific modifications these functions have undergone between versions.

Task 3: Code migration. The model is then required to revise or migrate the provided code to ensure it runs correctly in the target environment, addressing any version-related incompatibilities.

To evaluate CODEMENV, we conduct experiments with nine LLMs. Results show that, for the migration task, the average pass@1 rate across seven models is 26.50%, with GPT-4O achieving the highest score at 43.84%. Our analysis uncovers several notable insights:

(i) Familiarity with function versions. LLMs tend to be more familiar with newer function versions, which enables them to migrate legacy code to modern environments more effectively, but makes adapting newer code to older environments more challenging.

(ii) Logical inconsistencies. LLMs sometimes display logical inconsistencies when identifying relevant function changes for migration. For example, when migrating code from version 1.16 to 2.0, models may mistakenly reference changes from version 1.0, which are not pertinent to the migration at hand.

2 Related Work

2.1 LLMs for Code

Large Language Models (LLMs) have achieved remarkable progress in code-related tasks such as generation, translation, and completion, owing to their large parameter counts and training on vast code corpora. Proprietary models like GPT-4 (Achiam et al., 2023), Claude-3 (The), and Gemini (Reid et al., 2024) consistently deliver strong performance across a broad spectrum of programming challenges. In parallel, open-source models such as Qwen2.5 (Team, 2024) have demonstrated competitive or even superior results compared to larger models, leveraging synthetic data to enhance their capabilities. Other open-source models, including Llama-3.1 (Abhimanyu Dubey et al., 2024), Phi-3 (Abdin et al., 2024), and DeepSeek (Shao et al., 2024), also achieve impressive results, with DeepSeek in particular outperforming many proprietary alternatives.

The field has also seen rapid advances in specialized CodeLLMs. For example, CodeT5 (Wang et al., 2021) employs an encoder-decoder architecture and excels at code completion and retrieval, while CodeT5+ (Wang et al., 2023) introduces flexible encoder-only, decoder-only, and encoder-decoder modes, achieving strong results on benchmarks like HumanEval. CodeLlama (Rozière et al., 2023), developed by Meta, extends Llama 2 with code-specific training and supports multiple languages, with its 34B variant showing robust performance in code generation and completion. StarCoder2 (Lozhkov et al., 2024), from BigCode, is designed for multi-language code synthesis and retrieval, leveraging large-scale permissively licensed datasets. Qwen-Coder (Hui et al., 2024) is notable for its 32B variant, which surpasses GPT-4o on several benchmarks, benefiting from training on 5.5T tokens of diverse data and strong human preference alignment. These developments underscore the rapid evolution of domain-specific LLMs and the narrowing gap between open-source and proprietary solutions.

In this work, we assess the LLMs' ability to migrate code across different environments.

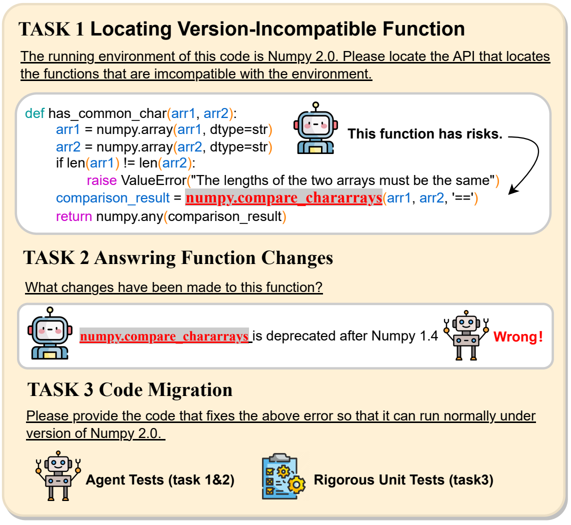
2.2 Code Migration

Recent progress in AI-driven code migration has demonstrated encouraging results across diverse programming scenarios. Amazon Q Developer (AmazonQ) exemplifies a generative AI tool tailored to assist developers in upgrading Java applications from versions 8/11 to 17, addressing the broader challenges of repository-level code migration. Joe (Khoury, 2024) provides a comprehensive analysis of the current landscape and persistent obstacles in large-scale migration efforts. The dynamics of human-AI collaboration in this context are explored by Omidvar Tehrani et al. (Tehrani et al., 2024), who assess how developers and Amazon Q Developer interact during migration tasks. Google researchers (Ziftci et al., 2025) have introduced an automated approach that integrates change location discovery with LLM-based guidance to streamline migration processes. In the domain of legacy system modernization, Kontogiannis (Kontogiannis et al., 2010) proposes a semi-automated method for migrating PL/IX to C++, emphasizing iterative optimization to address performance issues. More recently, Almeida et al. (Almeida et al., 2024) demonstrated that ChatGPT can effectively support API migration, with One-Shot prompting proving particularly effective for migrating Python/SQLAlchemy applications while preserving functionality and adopting modern features.

Additional discussion of related work is provided in Appendix A.

3 CodeMEnv

We argue that despite significant challenges that environment-related issues present to programmers, there is currently no systematic benchmark for evaluating model capabilities in code migration across different environments. To fill this gap, we introduce CodeMEnv (**Code** **M**igration Across **Env**ironments), a benchmark designed to assess models' understanding of function usage differences across library versions and their ability to perform cross-version code migration. The remainder of this section details the construction and characteristics of CodeMEnv.


TASK 1 Locating Version-Incompatible Function

The running environment of this code is Numpy 2.0. Please locate the API that locates the functions that are imcompatible with the environment.

```
def has_common_char(arr1, arr2):
    arr1 = numpy.array(arr1, dtype=str)
    arr2 = numpy.array(arr2, dtype=str)
    if len(arr1) != len(arr2):
        raise ValueError("The lengths of the two arrays must be the same")
    comparison_result = numpy.compare_chararrays(arr1, arr2, '==')
    return numpy.any(comparison_result)
```

This function has risks.

TASK 2 Answring Function Changes

What changes have been made to this function?

numpy.compare_chararrays is deprecated after Numpy 1.4

Wrong!

TASK 3 Code Migration

Please provide the code that fixes the above error so that it can run normally under version of Numpy 2.0.

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

3.1 Task Definition

CodeMEnv include the following three tasks:

Task-1: Identify Version-Incompatible Functions. Given a code snippet and a specified target environment version, the model is asked to pinpoint functions that are incompatible with that environment. This task is divided into two levels of difficulty: **easy**, where only one incompatible function is present, and **hard**, where multiple incompatible functions must be identified.

Task-2: Answering Function Changes. For each function identified in Task-1 the model must explain the nature of the change. This includes specifying the type of change (e.g. deprecation), the version in which the change occurred, and any replacement function if applicable.

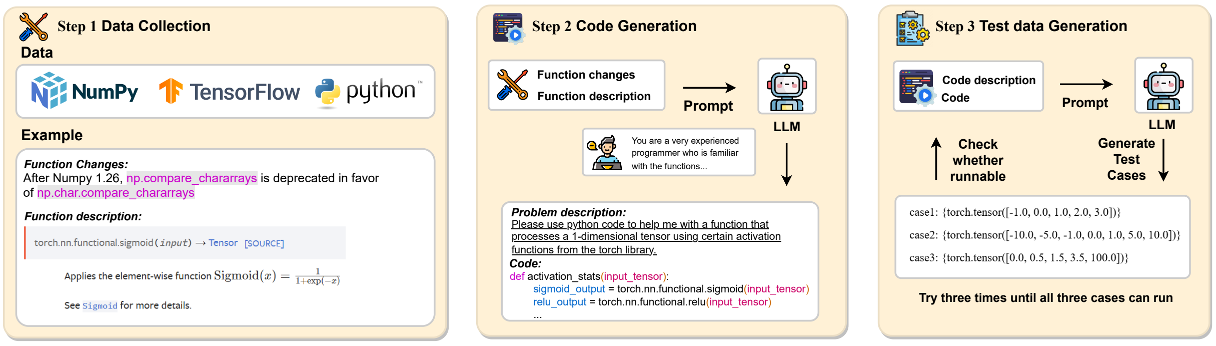
Task-3: Code Migration. The model is required to revise the provided code so that it is compatible with the target environment. Code migration scenarios are further categorized as follows:

(a) New2Old: The target environment version is older than the original, so the model must adapt newer code to run in a legacy environment.

(b) Old2New: The target environment version is newer than the original, requiring the model to upgrade legacy code for compatibility with the updated environment.

3.2 Dataset Statistics

CodeMEnv encompasses two widely used programming languages in the deep learning community: Python and Java. The Python portion of the dataset spans 11 packages and contains a to-



Function Changes:
After Numpy 1.26, np.compare_chararrays is deprecated in favor of np.char.compare_chararrays

Function description:
torch.nn.functional.sigmoid(*input*) → Tensor [SOURCE]

Applies the element-wise function $\text{Sigmoid}(x) = \frac{1}{1+\exp(-x)}$

See `Sigmoid` for more details.

You are a very experienced programmer who is familiar with the functions...

Problem description:
Please use python code to help me with a function that processes a 1-dimensional tensor using certain activation functions from the torch library.
Code:
```
def activation_stats(input_tensor):
    sigmoid_output = torch.nn.functional.sigmoid(input_tensor)
    relu_output = torch.nn.functional.relu(input_tensor)
    ...
```

```
case1: {torch.tensor([-1.0, 0.0, 1.0, 2.0, 3.0])}
case2: {torch.tensor([-10.0, -5.0, -1.0, 0.0, 1.0, 5.0, 10.0])}
case3: {torch.tensor([0.0, 0.5, 1.5, 3.5, 100.0])}
```

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.

tal of 587 samples, which are divided into two difficulty levels: **easy** and **hard**. The **easy** subset comprises 396 samples, each featuring a single line of code that is incompatible with the target environment. The **hard** subset includes 191 samples, each containing k incompatible lines of code, where $k \in \{2, 3\}$. Table 1 summarizes the distribution of incompatible lines across the datasets.

For Java, the dataset covers 8 packages with 335 samples. Only the **easy** difficulty level is included for Java, as the incompatible functions in these packages tend to be loosely connected, making it difficult to construct **hard** instances with multiple interdependent incompatibilities.

Additional details and comprehensive statistics for CodeMEnv are provided in Appendix C.1.

3.3 Function Changes

We categorize function changes in CodeMEnv into three main types:

- **Addition** (`None` $\rightarrow f_{\text{new}}$): A new function f_{new} is introduced in a later version, meaning it is unavailable in earlier environments.
- **Deprecation** ($f_{\text{old}} \rightarrow$ `None`): The function f_{old} is removed or no longer supported after a certain version, so it cannot be used in newer environments.
- **Replacement** ($f \rightarrow f'$): The function f is replaced or modified to f', which may involve changes to the function name, parameters, or usage patterns.

Table 6 in Appendix C.1 summarizes the distribution of these change types in the Python portion of CodeMEnv: 98 replacements, 35 deprecations, and 79 additions.

Datasets	1-incom.	2-incom.	3-incom.	Total
Python (easy)	396	-	-	396
Python (hard)	-	103	88	191
Java	335	-	-	335

Table 1: Statistics of the number of incompatible lines of CodeMEnv.

3.4 Evaluation

In this section, we outline the evaluation methodology for the three benchmark tasks, utilizing two primary approaches:

Agent-based Evaluation. To evaluate whether LLMs can accurately identify version-incompatible functions and correctly describe the changes those functions have undergone, we adopt an agent-based evaluation strategy. The agent is provided with ground-truth answers, including the set of incompatible functions and their corresponding changes. It then compares the LLMs' predictions to these references according to the following criteria:

For Task-1, the evaluation requires the model to identify all functions that are incompatible with the target environment. The prediction is considered correct only if the set of identified functions exactly matches the ground truth; missing or incorrectly including any function results in failure. The accuracy for Task-1 is defined as:

$$\text{Acc}_{\text{Task-1}} = \mathbb{1}\left[\mathcal{I} = \mathcal{T}\right] = \begin{cases} 1, & \text{if } \mathcal{I} = \mathcal{T} \\ 0, & \text{otherwise} \end{cases}, \quad (1)$$

where $\mathcal{I}$ is the set of ground-truth incompatible functions, and $\mathcal{T}$ is the set predicted by the LLM.

For Task-2, we assess three aspects: (i) whether the LLM correctly identifies the type of change (see Section 3.3); (ii) whether the predicted version number of the change is accurate (allowing a margin of 0.5 between predicted and actual version numbers); (iii) for replacement-type changes, whether the LLM correctly specifies the replacement function (this is not required for addition or deprecation cases). The accuracy for Task-2 is given by:

$$\mathrm{Acc}_{\text{Task-2}} = \mathbb{1}[\, \underbrace{\hat{t} = t}_{\text{type}} \wedge \underbrace{|\hat{v} - v| \leq 0.5}_{\text{version}} \wedge \underbrace{(t \neq \text{`Replace'} \vee \hat{f} = f)}_{\text{function}}], \tag{2}$$

where $\hat{t}$ and t are the predicted and ground-truth change types, $\hat{v}$ and v are the predicted and actual version numbers, and $\hat{f}$ and f are the predicted and ground-truth replacement functions.

Further details on the agent-based evaluation are provided in the third prompt of Appendix B.

Unit Test-based Evaluation. For Task-3, we assess whether the migrated code preserves the original functionality by running a set of test cases on both the original and migrated implementations.

Specifically, three test cases are executed for each code pair. The outputs of the original code serve as the reference, and the migrated code is considered correct only if it produces identical outputs for all test cases. The accuracy for Task-3 is defined as:

$$\mathrm{Acc}_{\text{Task-3}} = \mathbb{1}\left[\bigwedge_{i=1}^{3}(m_i = o_i)\right], \tag{3}$$

where m_i is the output of the migrated code and o_i is the output of the original code for the i-th test case.

3.5 Construction Process

Figure 3 presents the data curation workflow for CODEMENV, which comprises three stages:

Step 1: Data Collection. We begin by gathering a comprehensive set of functions, along with their associated changes, descriptions, and supported version ranges.

To achieve this, we systematically review version release notes from the official documentation of each package, cataloging all modified functions and detailing their changes across different versions. We also extract functional descriptions and usage information for each function to support subsequent code generation. Since official documentation often omits explicit version compatibility information, we empirically determine the supported version ranges by executing the functions across multiple package versions.

In total, our analysis yields 212 function changes for Python and 114 for Java. The online sources referenced for this collection are listed in Appendix C.2.

Step 2: Code Generation. Next, we generate original code samples based on the collected data. This step leverages the advanced capabilities of GPT-4: by providing it with the function changes, original function definitions, and usage descriptions, we prompt GPT-4 to generate code that correctly utilizes these functions.

The code generation scenario depends on the type of function change (*i.e.*, OLD2NEW or NEW2OLD):

(i) For addition-type changes (`None` $\rightarrow f_{\text{new}}$), we create NEW2OLD samples. GPT-4 is given the newly introduced function f_{new} and asked to generate code compatible with the newer environment where f_{new} exists. The migration target is the version prior to the change, where f_{new} is unavailable.

(ii) For deprecation-type changes ($f_{\text{old}} \rightarrow$ `None`), we create OLD2NEW samples. GPT-4 receives the deprecated function f_{old} and generates code that runs in the older environment where f_{old} is still present. The migration target is the version after the change, where f_{old} has been removed.

(iii) For replacement-type changes ($f \rightarrow f'$), we generate both OLD2NEW and NEW2OLD samples. GPT-4 is prompted separately with each function to produce the corresponding code samples for both migration directions.

Further details on this process are provided in the first and sixth prompts of Appendix B.

Step 3: Test Case Generation. Finally, we construct test cases for each generated code sample to ensure that migrated code preserves functional correctness.

For this, GPT-4 is supplied with both the original code and its functional specification and instructed to generate three test cases. These are executed on the original code to obtain ground-truth outputs, which are then used in Task-3 to assess the correctness of migrated code.

Occasionally, generated test cases may exhibit issues such as invalid input ranges, incorrect formats, or runtime errors, which may arise from

Base Model	Task 1 Locating Function				Task 2 Answering Change			
	Python (easy)	Python (hard)	Java	Avg.	Python (easy)	Python (hard)	Java	Avg.
GPT-TURBO-3.5	**85.10**	**32.98**	80.89	**66.32**	<u>26.01</u>	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	<u>13.61</u>	**75.22**	<u>37.02</u>
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	<u>29.84</u>	81.19	62.18	22.73	8.38	**75.22**	35.44
DEEPSEEK-V3	<u>78.48</u>	26.17	<u>82.08</u>	<u>62.24</u>	**38.99**	**16.75**	<u>70.44</u>	**42.06**

Table 2: Experiment results for Task-1 and Task-2. We **bold** the best result and <u>underline</u> the second-best result. The first two tasks only test general LLMs, because code-specialized LLMs focus more on generating code.

either the test cases themselves or defects in the original code. To address this, we iteratively provide GPT-4 with error messages from failed executions, allowing it to refine the test cases. This refinement process is repeated for up to three iterations. If all three test cases execute successfully, the data sample is retained; otherwise, both the test cases and the associated code are discarded.

Details of this step are provided in the fourth prompt of Appendix B.

4 Experimentation

In this section, we present a comprehensive analysis of our experimental setup and results.

4.1 Experimental Settings

Large Models. We conduct experiments on nine different LLMs. These include six general LLMs, namely: GPT-TURBO-3.5 (Ye et al., 2023), GPT-4O-MINI (OpenAI et al., 2024a), GPT-4O (OpenAI et al., 2024b), LLAMA-3.1-8B-INSTRUCT (Abhimanyu Dubey et al., 2024), LLAMA-3.1-70B (Abhimanyu Dubey et al., 2024), and DEEPSEEK-V3 (Shao et al., 2024); three code-specialized LLMs: QWEN2.5-CODER-7B-INSTRUCT (Hui et al., 2024), STARCODER2-15B (Lozhkov et al., 2024), and CODE LLAMA-34B (Rozière et al., 2023).

Evaluation Metrics. We assess model performance on the three tasks using the following metrics: $\text{Acc}_{\text{Task\_1}}$, $\text{Acc}_{\text{Task\_2}}$, and $\text{Acc}_{\text{Task\_3}}$, as detailed in Section 3.4.

For Task-3 (code migration), we further report the Pass@k metric (Hendrycks et al., 2021), which quantifies the proportion of examples for which the model produces at least one correct migration within k attempts. Formally,

$$\text{Pass@}k = \mathbb{I}\left[\bigcup_{i=1}^{k} \text{Acc}_{\text{Task\_3}}^{(i)}\right] \qquad (4)$$

where $\text{Acc}_{\text{Task\_3}}^{(i)}$ is an indicator of whether the i-th attempt for Task-3 is successful.

Experiment Setup. For all LLMs, we set the generation temperature to 0.7 and limit the maximum output sequence length to 2048 tokens. Proprietary models (e.g., the GPT series) are evaluated via their official APIs. For smaller open-source models such as QWEN2.5-CODER-7B-INSTRUCT, we run inference locally using two RTX 4090 GPUs. For large-scale open-source models, *i.e.,* LLAMA-3.1-70B, we access them through APIs provided by third-party websites [†].

4.2 Main Experiments

Table 2 summarizes the experimental results for Task-1 and Task-2. Below, we discuss the key findings:

Overall Performance of Task-1. The average locating success rate ($\text{Acc}_{\text{Task\_1}}$) for the six general LLMs across both Python and Java is 59.76%. Performance is notably strong on the Python (easy) and Java datasets, with average scores of 74.84% and 79.53%, respectively. However, all models struggle on the Python (hard) dataset, which contains multiple incompatible functions per example. For instance, QWEN2.5-CODER-7B achieves only a 15.71% pass rate in this setting. This drop in performance is primarily due to the models' difficulty in identifying all incompatible functions: when several such functions are present, models often detect only a subset or make incorrect identifications.

Overall, GPT-TURBO-3.5 achieves the highest overall success rate on Task-1, with an average of 66.32%. Its strength is particularly apparent on the Python (hard) dataset, where it attains a 32.98% pass rate—substantially higher than other models. This suggests that GPT-TURBO-3.5 is more adept at comprehensively locating all version-incompatible functions in complex scenarios. While other models may overlook or misclassify some incompatible functions when multiple are present, GPT-TURBO-3.5 more consis-

[†]`https://cloud.siliconflow.cn/models`

Base Model	Task 3 Migration (OLD2NEW)				Task 3 Migration (NEW2OLD)			
	Python (easy)		Python (hard)		Python (easy)		Python (hard)	
	Pass@1	Pass@5	Pass@1	Pass@5	Pass@1	Pass@5	Pass@1	Pass@5
GENERAL LARGE LANGUAGE MODEL								
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
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
DEEPSEEK-V3	41.20	54.11	20.73	29.27	29.60	39.60	14.68	23.85
GPT-4O	**43.84**	**59.59**	**26.83**	**47.56**	**31.60**	43.60	**22.94**	**27.52**
CODE-SPECIALIZED LARGE LANGUAGE MODEL								
QWEN2.5-CODER-7B	32.19	46.58	14.63	24.39	29.20	38.00	8.26	12.84
STARCODER2-15B	32.19	46.58	12.54	28.73	28.80	38.40	13.50	18.35
CODE LLAMA-34B	**35.62**	**53.42**	**21.95**	**36.49**	**29.60**	**40.80**	**15.76**	**21.10**

Table 3: Experiment results for Task-3, we report the results in two cases: OLD2NEW and NEW2OLD.

tently identifies a greater proportion, leading to its superior performance. Nevertheless, it does not achieve the top results on Java, which may reflect differences in model proficiency across different programming languages.

Overall Performance of Task-2. The average success rate ($\text{Acc}_{\text{Task\_2}}$) for the six general LLMs across Python and Java is 33.96%, which is notably lower than their performance on Task-1. This gap highlights a key limitation: while LLMs can often identify version-incompatible functions, they struggle to recall or reason about the specific details of how those functions have changed across versions. In other words, LLMs are less adept at providing precise, contextually accurate descriptions of function modifications, replacements, or deprecations.

Among all models, DEEPSEEK-V3 stands out with the highest average score of 42.06%. Its advantage is particularly evident on the Python (easy) dataset, where it achieves 38.99%. A closer look at the scoring criteria for $\text{Acc}_{\text{Task\_2}}$ (see Section 3.4) reveals that DEEPSEEK-V3's strength lies in its ability to accurately recall the specific version in which a function change occurred—a capability that most other LLMs lack. This suggests that DEEPSEEK-V3 has either been exposed to more up-to-date or detailed training data, or possesses better mechanisms for temporal reasoning about software evolution.

Conversely, LLAMA-3.1-8B lags behind, with an average score of only 23.79%. Its primary weakness is in identifying replacement-type changes: it often fails to specify which function an incompatible one should be replaced with in the target environment. This indicates that smaller or less specialized models may lack the depth of codebase knowledge or the reasoning ability required for nuanced migration scenarios.

Overall Performance of Task-3. Table 3 presents the results for Task-3 (code migration). In the OLD2NEW scenario, the average Pass@1 success rate for the nine LLMs is 33.56% on the easy set and drops to 16.20% on the hard set. Notably, allowing more attempts substantially boosts performance: Pass@5 rises to 45.5% (easy) and 26.47% (hard), indicating that LLMs can often generate a correct migration with additional tries, even if their first attempt fails.

In contrast, the NEW2OLD scenario proves much more challenging. Here, the average Pass@1 and Pass@5 rates are only 12.77% and 17.30% (hard set), and additional attempts yield little improvement. This asymmetry suggests that LLMs are more familiar with migrating legacy code to newer environments than the reverse, likely reflecting the distribution of code and documentation in their training data.

Among all models, GPT-4O delivers the strongest performance in the OLD2NEW migration task, achieving a Pass@1 rate of 43.84% and a Pass@5 rate of 59.59% on the easy set. This demonstrates its superior ability to synthesize and adapt code for modern environments, likely due to its larger context window, more recent training data, and advanced reasoning capabilities. In contrast, GPT-TURBO-3.5, which excels at locating incompatible functions (Task-1), does not translate this strength into effective code migration: its Pass@1 rate on the hard set is only 7.32%. This discrepancy highlights that the skills required for identifying incompatibilities and for

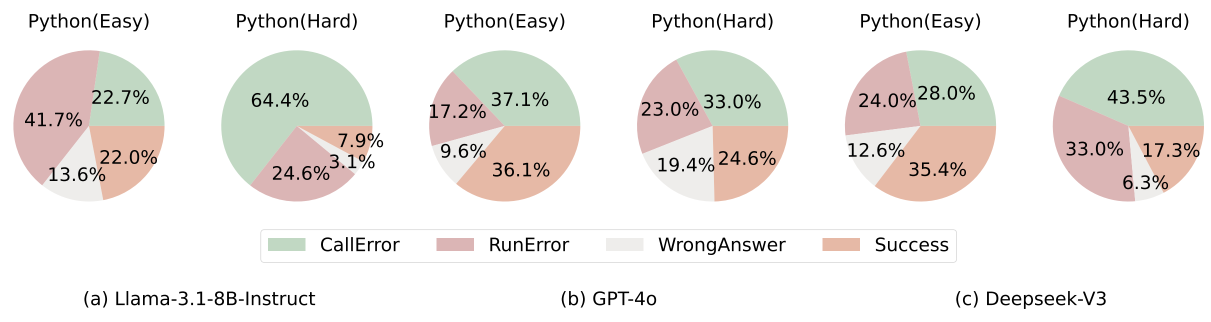


Figure 4: Error Analysis of Code Migration. `CallError` represents a function where an incompatible the environment is still called. `RunError` represents that the migrated code enters an infinite loop during execution. `WrongAnswer` represents this code runs normally and gets the result, but it is different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

generating correct, environment-adapted code are distinct. While GPT-TURBO-3.5 can assist users in pinpointing problematic functions, it is less reliable for fully automated migration, especially in complex scenarios.

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 training 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.

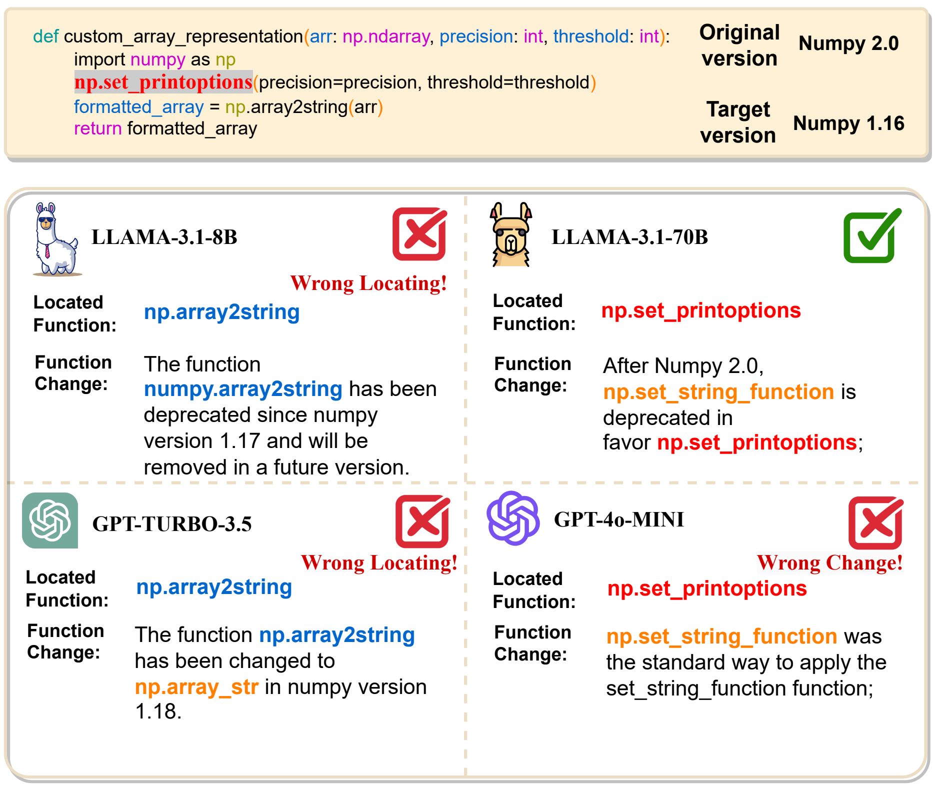
Error Analysis for Code Migration. To better understand the types of errors that occur during code migration, we conduct an error analysis of the failed cases, as illustrated in Figure 4. We categorize the failures into several types. The most prevalent is `CallError`, which arises when the generated code still invokes a function that is incompatible with the target environment. For example, 50.8% of the code produced by LLAMA-3.1-8B for the Python (**hard**) migration task fails due to this error. Such failures can occur either because the model does not successfully identify all incompatible functions, or because, even after correctly locating them, it still generates code that calls an incompatible function. Another common error type is `RunError`, where the code compiles and runs but enters an infinite loop or otherwise fails to terminate in a reasonable time. For instance, 33.0% of the code generated by DEEPSEEK-CHAT failed due to this issue.

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. The goal of this case study (Figure 5) is to analyze how different LLMs perform on the tasks of locating version-incompatible functions and accurately describing their changes, with a focus on their reasoning about version constraints.

Our findings reveal two main types of errors. First, both LLAMA-3.1-8B and GPT-TURBO-3.5 fail to correctly identify the relevant incompatible function. Instead, they focus on `np.array2string`, providing information about changes introduced in NumPy versions 1.17 and 1.18, even though the target environment is 1.16. Since these changes do not impact functionality in version 1.16, the models' responses are irrelevant for the migration task. This suggests a common failure mode: incorrect reasoning about version ordering, where models conflate changes from later versions with the requirements of an earlier target environment.

In contrast, LLAMA-3.1-70B and GPT-4O-MINI correctly identify `np.set_printoptions` as incompatible with NumPy 1.16. However, GPT-4O-MINI struggles to specify the precise version in which the function change occurred, providing inaccurate version information. This issue—misreporting the



```
def custom_array_representation(arr: np.ndarray, precision: int, threshold: int):
    import numpy as np
    np.set_printoptions(precision=precision, threshold=threshold)
    formatted_array = np.array2string(arr)
    return formatted_array
```

LLAMA-3.1-8B — Wrong Locating!

Located Function: np.array2string

Function Change: The function numpy.array2string has been deprecated since numpy version 1.17 and will be removed in a future version.

LLAMA-3.1-70B

Located Function: np.set_printoptions

Function Change: After Numpy 2.0, np.set_string_function is deprecated in favor np.set_printoptions;

GPT-TURBO-3.5 — Wrong Locating!

Located Function: np.array2string

Function Change: The function np.array2string has been changed to np.array_str in numpy version 1.18.

GPT-4o-MINI — Wrong Change!

Located Function: np.set_printoptions

Function Change: np.set_string_function was the standard way to apply the set_string_function function;

Figure 5: Case Study. We plot an example of NEW2OLD from Python (Easy) 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.

version associated with a function change—was frequently observed across our evaluation, highlighting a broader challenge for LLMs in tracking the evolution of library APIs with precision.

Overall, this case study demonstrates that even when models can identify relevant functions, they often fail in reasoning about version boundaries and providing accurate change details, which are critical for reliable code migration.

5 Conclusion

In this work, we introduced CODEMENV, a comprehensive benchmark designed to assess the code migration capabilities of LLMs across different environments. CODEMENV encompasses three core tasks: detecting version-incompatible functions, identifying specific function changes, and migrating code to ensure compatibility.

Our evaluation of nine LLMs demonstrates that models are generally more proficient with newer function versions, which poses difficulties for migrating code from newer to older environments (NEW2OLD). Additionally, our error analysis highlights logical inconsistencies, where the changes proposed by models do not always facilitate successful migration. We hope that CODEMENV and the insights from our experiments will inspire further research into improving LLM-driven code migration.

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.

Acknowledgements

This work is supported in part by the funding BAS/1/1689-01-01, URF/1/4663-01-01, REI/1/5232-01-01, REI/1/5332-01-01, and URF/1/5508-01-01 from KAUST, and funding from KAUST - Center of Excellence for Generative AI, under award number 5940.

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*.

Aylton Almeida, Laerte Xavier, and Marco Túlio Valente. 2024. Automatic library migration using large language models: First results. *Proceedings of the 18th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*.

AmazonQ. Amazon q developer: Transform code. 2025.

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.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Kai Dang, An Yang, Rui Men, Fei Huang, Shanghaoran Quan, Xingzhang Ren, Xuancheng Ren, Jingren Zhou, and Junyang Lin. 2024. Qwen2.5-coder technical report. *ArXiv*, abs/2409.12186.

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.

Joe El Khoury. 2024. Leveraging large language models for automated code migration and repository-level tasks — part i.

Kostas Kontogiannis, Johannes Martin, Kenny Wong, Richard Gregory, Hausi A. Müller, and John Mylopoulos. 2010. Code migration through transformations: an experience report. In *Conference of the Centre for Advanced Studies on Collaborative Research*.

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.

Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, Tianyang Liu, Max Tian, Denis Kocetkov, Arthur Zucker, Younes Belkada, Zijian Wang, and Qian Liu. 2024. Starcoder 2 and the stack v2: The next generation. *ArXiv*, abs/2402.19173.

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.

Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, I. Evtimov, Joanna Bitton, Manish P Bhatt, Cris tian Cantón Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre D'efossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. 2023. Code llama: Open foundation models for code. *ArXiv*, abs/2308.12950.

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.

Behrooz Omidvar Tehrani, Ishaani M, and Anmol Anubhai. 2024. Evaluating human-ai partnership for llm-based code migration. *Extended Abstracts of the CHI Conference on Human Factors in Computing Systems*.

Yue Wang, Hung Le, Akhilesh Deepak Gotmare, Nghi D. Q. Bui, Junnan Li, and Steven C. H. Hoi. 2023. Codet5+: Open code large language models for code understanding and generation. In *Conference on Empirical Methods in Natural Language Processing*.

Yue Wang, Weishi Wang, Shafiq R. Joty, and Steven C. H. Hoi. 2021. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *ArXiv*, abs/2109.00859.

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.

Zhuoran Zhang, Yongxiang Li, Zijian Kan, Keyuan Cheng, Lijie Hu, and Di Wang. 2024. Locate-then-edit for multi-hop factual recall under knowledge editing. *ArXiv*, abs/2410.06331.

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*.

Celal Ziftci, Stoyan Nikolov, Anna Sjovall, Bo Kim, Daniele Codecasa, and Max Kim. 2025. Migrating code at scale with llms at google. *ArXiv*, abs/2504.09691.

A Additional Related Work

A.1 Knowledge Editing

Knowledge editing is an effective way to add the latest knowledge of function changes to LLMs.

The research on knowledge editing for LLMs aims to efficiently modify large model's parameters in order 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 adopts a plan-and-solve paradigm (Zhong et al., 2023; Cheng et al., 2024), where complex problems are decomposed into the knowledge required for each step, which are then solved one by one. (Zhang et al., 2024) proposes a locate-then-edit paradigm to support efficient knowledge editing for multi-hop questions.

There are only a few research attempts 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 of the recent works (Zhou et al., 2022; Su et al., 2024; Hsieh et al., 2023) use retrieval-augmented approaches (Lewis et al., 2020; Guu et al., 2020) to provide models with code change knowledge for improving code generation.

Note, unlike existing work, CODEMENV does not supply the model with contextual knowledge of function changes during evaluation. Instead, we prioritize assessing how effectively the model leverages its inherent knowledge of function changes to perform code migration.

B Prompts for CODEMENV

See Prompt1 for the generation of original code of Python language (step-2 of datasets construction).

See Prompt2 for prompt we used in experiment to execute the three tasks of CODEMENV (Python).

See Prompt3 for the agent-based evaluation for Python language.

See Prompt4 for the generation of test cases.

See Prompt5 for the improving of test cases.

See Prompt6 for the generation of original code of Java language (step-2 of datasets construction).

See Prompt7 for prompt we used in experiment to execute the three tasks of CODEMENV (Java).

See Prompt8 for the agent-based evaluation for Java language.

Prompt 1. Original Code Generation: the Second Step for Dataset Construction of Python language

==== SYSTEM ====
You are a very experienced programmer who is familiar with the usage 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 description of this library function, be bold and creative, and then write a piece of code that calls this library function, we call this code as solution. This solution is a function, and 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, with the requirement that others should be able to derive a solution from this problem. Your problem description should focus on the solution's functionality, as well as its inputs and outputs, rather than guiding the step-by-step generation of the solution within the description
You must explicitly specify the data type and dimensionality of each input parameter, as well as the data type and parameters of the output. Your problem description should follow this template: "Please use Python code to implement a function..." Indicate which library is being utilized in the description, but refrain from specifying the exact library function being called. Avoid disclosing any implementation details.

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. The Prompt used by LLMs to Execute the Three Tasks of CodeMEnv (Python)

==== 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 contains functions that are incompatible with the target environment. You need to locate the incompatible function in this code. Then give the information about the change of this located function: (i) It must include the change type deprecation/addition/replacement; (ii) the replaced function(if the change type is replacement); (iii) and the version of this function that changed. Finally return your corrected code, and you only need to fix the incompatible function in the code.
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 function in the code because of the version,
ai_api_change: 1.The specified function(error function) 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 that the function 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. Agent-based Evluation: Task-1 and Task-2 of Python Language

==== SYSTEM ====
You are a good helper for a human being. I ask another LLM to locate the function in a piece of code that is incompatible with the environment, and its response include the following contents: (i) The located incompatible function ; (ii) information of the changes of the function; (iii) The migrated code after fixing the error.
...

Please compare the wrong functions returned by the AI and the correct function 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 function returned by the AI 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 absolutely are completely irrelevant, return 0. Remember 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. Test Cases Generation (Step-3 of Datasets Construction)

==== 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 scenario. 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 test cases quality

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

```
# Responsibilities
```
Adjust the test case group according to the provided executable script and running information, and return the adjusted test cases.

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

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

```
[/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 cases 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 cases extraction and identification
-Extract the parameters passed by the calling function from the executable script as the test cases group

Prompt 5: Improve test cases quality

2. Match the test cases group with the corresponding operation information
-Pair the test cases input groups in sequence according to the operation results
3. Save the test cases group that runs correctly and replace the test cases group that runs incorrectly
-Keep the test cases group that runs correctly unchanged
-For the test cases group that runs incorrectly, analyze the cause according to the error information, avoid similar errors, and replace them with new test case groups.
4. Finally, just return the modified test cases, 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: Original Code Generation: the Second Step for Dataset Construction of Java language

==== 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 function to complete a class, this function may have been removed in the new JDK. Assuming that I am running in an old JDK environment, please call the function anyway.

Then, generate a usage 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 usage description of your generated code.
}

==== USER ====
The signature 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: The Prompt used by LLMs to Execute the Three Tasks of CODEMENV (Java)

==== 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 a java JDK version mismatch. You need to locate the wrong functions in this code, and explain what version changes have taken place in the function 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: Agent-based Evluation: Task-1 and Task-2 of Java Language

==== SYSTEM ====
You are a good helper for a human being. I ask another LLM to locate the function in a piece of code that is incorrectly called because of the JDK version mismatch, and its response include the following contents: (i) The located incompatible function ; (ii) information of the changes of the function; (iii) The migrated code after fixing the error.

Please compare the wrong functions returned by the AI and the correct functions 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 function returned by the AI 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 absolutely are completely irrelevant, return 0. Remember 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]`

C CodeMEnv (Additional Details)

C.1 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 4: Statistics on the number of changes across different Java packages.

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

Table 5: Statistics on the number of changes across different Python packages.

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 6: Statistics of the number of three types of function changes across different packages of python language.

C.2 Data Collection Source

URL	**Description**
https://github.com/pytorch/pytorch/releases	Sources for collecting changes related to the PyTorch library.
https://numpy.org/doc/2.0/release/2.0.0-notes.html#changes	Sources for collecting changes related to the Numpy library.
https://docs.oracle.com/en/java/javase/11/docs/api/deprecated-list.html	Sources for collecting changes related to the Java library.
https://docs.python.org/zh-cn/3/library/random.html	Sources for collecting changes related to the random library.
https://github.com/tensorflow/tensorflow/releases/tag/v2.0.0	Sources for collecting changes related to the tensorflow library.
https://docs.python.org/zh-cn/3/library/itertools.html	Sources for collecting changes related to the itertools library.

Table 7: The URL for collecting data in step 1 of the data construction process.

C.3 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()
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.