The Role of Test Cases in LLM-Based Python and JavaScript Code Translation

Christy Zhang
christyyz16@gmail.com

Ethan McKeen
ethanrmckeen@gmail.com

Abstract

This project investigates the effectiveness of test cases in translating code between Python and JavaScript. We propose a pipeline that enhances translation accuracy through the use of large language models (LLMs) and iterative refinement. The pipeline begins by generating assertion-based synthetic test cases for the original code using an LLM, followed by an initial code translation prompt that includes an example translation pair and corresponding test cases. To evaluate translation correctness, we execute the translated code against ground truth test cases from the HumanEval-X dataset. If the initial translation fails, an iterative refinement step allows the LLM to improve upon previous attempts using both the original code and prior translation outputs. Experiments conducted with two LLMs—Gemini-2.0-Flash and GPT-3.5-Turbo—demonstrate that incorporating test cases significantly improves translation outcomes. Our findings suggest promising directions for enhancing code translation systems using test-aware prompting and structured model feedback.

1 Introduction

Large language models (LLMs) have shown impressive capabilities in code generation and understanding, making them promising tools for automatic code translation between programming languages. However, translating code between languages like Python and JavaScript remains a challenging task due to differences in syntax, semantics, and language-specific idioms. This project explores how the prompting strategy impacts the quality and correctness of code translation using LLMs. We investigate three distinct prompting methods: a baseline prompt consisting of plain English instructions and source code; a test-aware prompt that supplements the baseline with either ground truth or synthetic test cases; and a structured markdown-style prompt that includes an example translation pair with corresponding test cases to provide clearer context. Our goal is to evaluate how each prompt format influences the model's ability to produce accurate translations, and to analyze the role of test cases and in-context examples in improving translation performance. By conducting experiments in both translation directions (Python to JavaScript and vice versa), we aim to identify effective strategies for leveraging LLMs in real-world code conversion tasks.

2 Methodology

The general pipeline of this project is as follows. The user has some code in Python or JavaScript that they wish to convert to the other language. Our pipeline will first query the LLM API to generate test cases for this initial code. Next, it queries the LLM again to translate the code. This prompt will include an example JavaScript Python code translation pair with their ground truth test cases [1], along with the user's code and its newly generated test cases. If this initial translation is incorrect, an iterative refinement process can attempt to correct the translation.

[1] A ground truth test case refers to the human written test cases provided in the HumanEval-X dataset (Zheng et al. 2023).

2.1 Dataset

This project uses the Human-Eval-X dataset (Zheng et al. 2023) that contains source code in multiple programming languages along with corresponding test suites. The Python and JavaScript subsets are used to evaluate the correctness of bi-directional code translation between the two languages.

2.2 LLM Model Selection

Two LLM models are selected in our project: Gemini-2.0-Flash (Google Cloud 2024) and Gpt-3.5-Trubo-0125 (OpenAI 2023). Gemini-2.0-Flash is chosen because it was released on February 2025, which can represent current capabilities of LLMs in code translation. Gpt-3.5-Turbo is selected because it's knowledge cutoff is earlier than the release of our selected dataset. This ensures the results are not influenced by potential data memorization.

2.3 Generating Synthetic Test Cases

Synthetic test cases are generated using LLM models. For each function in the dataset, a prompt with instructions and the source language function code is created. The prompt asks the LLM to generate five assertion-based test cases. The original function implementation is used to evaluate the correctness of generated synthetic tests. Each test case was run against the original function and only those that pass are selected for use in our test-aware prompts during code translation. Test cases that take longer than five seconds to execute are also removed. To measure test effectiveness, we use pytest for Python and nyc for JavaScript. Since the functions in HumanEval-X are relatively simple, we choose to focus on statement coverage.

2.4 Code Translation

The project use LLM to translate code between Python and JavaScript languages. Three sets of experiments will be done to assess whether test cases can improve translation correctness. For each function in the dataset, we prompt the LLM three times to translate. The baseline prompt includes only instructions and source language function code, while the test-aware prompts includes additional ground truth tests or synthetic tests. An experiment with single-shot prompt is also conducted.

To evaluate translation success, we execute the translated code against the original ground truth test cases in the source language. A translation is considered successful if the translated code passes all the provided ground truth tests.

2.5 Iterative Refinement

The final step of the pipeline involves an iterative refinement process applied to code translations that initially fail. In this step, the LLM is queried up to five times in succession. Each iteration provides the LLM with the previously generated, incorrect translation along with the original source code, allowing it to incrementally improve upon its last attempt. This feedback loop is designed to progressively correct translation errors by leveraging the model's ability to reflect on and refine its own outputs, ultimately increasing the likelihood of producing a syntactically and semantically correct translation.

3 Results

3.1 Synthetic Test Pass Rate and Coverage

Table 1 presents the test case pass distributions for synthetic tests generated by the gemini-2.0-flash and gpt-3.5-turbo-0125 models.

Table 1: Number of functions grouped by how many LLM-generated test cases passed against original implementations from HumanEval-X dataset. Test cases were generated by `gemini-2.0-flash` and `gpt-3.5-turbo-0125`. Percentages are out of 164 total functions.

# Passed	Gemini-2.0-Flash		GPT-3.5-Turbo-0125	
	Python	JavaScript	Python	JavaScript
5	96 (58.54%)	96 (58.54%)	65 (39.63%)	55 (33.54%)
4	27 (16.46%)	30 (18.29%)	32 (19.51%)	34 (20.73%)
3	25 (15.24%)	19 (11.59%)	24 (14.63%)	20 (12.20%)
2	7 (4.27%)	6 (3.66%)	16 (9.76%)	21 (12.80%)
1	6 (3.66%)	6 (3.66%)	13 (7.93%)	17 (10.37%)
0	3 (1.83%)	7 (4.27%)	14 (8.54%)	17 (10.37%)

Table 2 presents the statement coverage of synthetic tests generated using both models. Coverage of ground truth test cases are also included for reference.

Table 2: Test Coverage for Synthetic and Ground Truth Tests

Test Type	Model	Python Test Coverage	JavaScript Test Coverage
Synthetic Test	Gemini-2.0-Flash	95.73%	92.94%
	GPT-3.5-Turbo-0125	89.02%	84.65%
Ground Truth Test	-	99.44%	99.48%

3.2 Code Translation with Prompt Engineering

LLMs are sensitive to the prompt given when completing a task. Different prompt can give vastly different results so multiple tests with varying prompt formats were completed to find the optimal prompting method. These methods include baseline, test-aware, and markdown single shot prompting. The use of markdown and few shot prompting was inspired by the results found by the LIBRO model (Kang, Yoon, and Yoo 2022).

3.2.1 Baseline Prompt

The first prompting method was a baseline, plain english prompt where the task of translating Python to JavaScript (or the other way around) was defined and the source code is provided. No test cases were given in the prompt and iterative refinement was not applied. An example of this type of prompt can be seen in appendix A.1. Rather than just prompting the LLM to translate the code a more complicated prompt proved beneficial where the LLM was reminded to preserve a language's conventions such as camel case for JavaScript and snake case for Python. To evaluate this prompting method, the Gemini LLM was queried for each example in the Human-Eval-X dataset. There are 164 Python-JavaScript code pairs in the dataset so the LLM was queried 164 times per direction. It was found that, with this baseline prompt, 87.8% of the examples were successfully translated into their target language by Gemini-2.0-Flash going in both directions; by GPT-3.5-Turbo-0125, 42.07% in Python to JavaScript and 46.59% in JavaScript to Python are successfully translated, as shown in Table 3.

Table 3: Baseline Prompt Pass Rate for Gemini-2.0-Flash and GPT-3.5-Turbo-0125

Model	Direction	Passes	Total	Accuracy
Gemini-2.0-Flash	Python $\rightarrow$ JavaScript	144	164	87.80%
	JavaScript $\rightarrow$ Python	144	164	87.80%
GPT-3.5-Turbo-0125	Python $\rightarrow$ JavaScript	69	164	42.07%
	JavaScript $\rightarrow$ Python	77	164	46.59%

3.2.2 Test-Aware Prompt

The second prompting method was a test-aware prompt. The prompt was again formatted in plain english, the task of translating code was defined, and the source code is provided. Contrary to the baseline, this prompt included test cases in the language of the source code but iterative refinement was still not applied. An example of this type of prompt can be seen in appendix A.2. Two tests were performed under this prompting method; one where the provided test cases were ground truth and one where the provided test cases were synthetically generated from previous LLM queries.

Ground Truth: To evaluate this prompting method, the LLM was queried for each example in the Human-Eval-X dataset. It was found that providing the LLM with test cases improved the performance in both models. This is expected because the test cases provide the LLM with extra insight into the intended function behaviors. The results can be seen in Table 4.

Table 4: Test-Aware Prompt Pass Rate for Gemini-2.0-Flash and GPT-3.5-Turbo-0125 (Ground Truth Tests)

Model	Direction	Passes	Total	Accuracy
Gemini-2.0-Flash	Python → JavaScript	146	164	89.02%
	JavaScript → Python	149	164	90.85%
GPT-3.5-Turbo-0125	Python → JavaScript	101	164	61.59%
	JavaScript → Python	115	164	70.12%

Synthetic: It was found that providing the LLM with synthetic test cases also improved the performance compared to the baseline, but the performance is similar or worse than using ground truth test cases. This is expected considering the quality of synthetic test cases compared to ground truth tests is lower as seen in the synthetic test case pass rate and coverage results. The pass rates for test-aware prompting can be seen in Table 5.

Table 5: Test-Aware Prompt Pass Rate for Gemini-2.0-Flash and GPT-3.5-Turbo-0125 (Synthetic Tests)

Model	Direction	Passes	Total	Accuracy
Gemini-2.0-Flash	Python → JavaScript	149	164	90.85%
	JavaScript → Python	146	164	89.02%
GPT-3.5-Turbo-0125	Python → JavaScript	80	164	48.78%
	JavaScript → Python	105	164	64.02%

3.2.3 Markdown and Single Shot Prompt

The final prompting method formatted the prompt as a markdown and implemented single shot testing. In the prompt, the task of translating code was defined, and the source code was provided in a markdown code block indicated by ’“‘’. This prompt included test cases in the language of the source code but iterative refinement was still not applied. An example of this type of prompt can be seen in appendix A.3. Again two tests were performed under this prompting method; one where the provided test cases were ground truth and one where the provided test cases were synthetically generated from previous LLM queries. It was found that giving the prompt more structure by formatting it as a markdown and providing single shot test examples improved the translation capability.

Ground Truth: Performance improved compared to baseline and plain english prompts particularly for GPT-3.5. The results can be seen in Table 6.

Table 6: Markdown and Single Shot Prompt Pass Rate for Gemini-2.0-Flash and GPT-3.5-Turbo-0125 (Ground Truth Tests)

Model	Direction	Passes	Total	Accuracy
Gemini-2.0-Flash	Python → JavaScript	162	164	98.78%
	JavaScript → Python	148	164	90.24%
GPT-3.5-Turbo-0125	Python → JavaScript	130	164	79.27%
	JavaScript → Python	137	164	83.54%

Synthetic: Performance generally improved compared to baseline and plain english prompts except for Gemini when converting JavaScript to Python. The translation ability is very comparable to the previous results when ground truth tests were used. The results can be seen in Table 7.

Table 7: Markdown and Single Shot Prompt Pass Rate for Gemini-2.0-Flash and GPT-3.5-Turbo-0125 (Synthetic Tests)

Model	Direction	Passes	Total	Accuracy
Gemini-2.0-Flash	Python → JavaScript	155	164	94.51%
	JavaScript → Python	145	164	88.41%
GPT-3.5-Turbo-0125	Python → JavaScript	130	164	79.27%
	JavaScript → Python	137	164	83.54%

3.3 Iterative Refinement

To further improve the translation ability and fix the remaining code examples that could not translate, an iterative step was added onto the end of the pipeline. This process involved retrieving the erroneous translated code and re-querying the LLM providing the initial source code and the erroneous translation in the prompt. Given the performance increased previously observed by prompt engineering, the prompt of the re-query was also formatted as a markdown.

Ground Truth Tests: The first test used the incorrect code translations previously generated from prompts in which ground truth test cases were provided. After one iteration of iterative refinement the results seen in Table 8 were observed. Notably, Gemini was able to successfully translate all of the code examples in the Python to JavaScript direction. After five iterations the results were further improved as seen in Table 9.

Table 8: One Iteration of Iterative Refinement Pass Rate (Ground Truth Tests)

Model	Direction	Passes	Total	Accuracy
Gemini-2.0-Flash	Python → JavaScript	164	164	100.00%
	JavaScript → Python	160	164	97.56%
GPT-3.5-Turbo-0125	Python → JavaScript	148	164	90.24%
	JavaScript → Python	142	164	86.59%

Table 9: Five Iterations of Iterative Refinement Pass Rate (Ground Truth Tests)

Model	Direction	Passes	Total	Accuracy
Gemini-2.0-Flash	Python → JavaScript	164	164	100.00%
	JavaScript → Python	162	164	98.78%
GPT-3.5-Turbo-0125	Python → JavaScript	151	164	92.07%
	JavaScript → Python	146	164	89.02%

Synthetic Tests: The second test used the incorrect code translations previously generated from prompts in which synthetic test cases were provided. After one and five iterations of iterative refinement the results seen in Tables 10 and 11 were observed respectively. Gemini was again able to successfully translate all of the code examples in the Python to JavaScript direction but struggled more in the opposite direction. The synthetic test results are still very comparable with the ground truth test case results.

Table 10: One Iteration of Iterative Refinement Pass Rate (Synthetic Tests)

Model	Direction	Passes	Total	Accuracy
Gemini-2.0-Flash	Python → JavaScript	164	164	100.00%
	JavaScript → Python	147	164	89.63%
GPT-3.5-Turbo-0125	Python → JavaScript	148	164	90.24%
	JavaScript → Python	142	164	86.59%

Table 11: Five Iterations of Iterative Refinement Pass Rate (Synthetic Tests)

Model	Direction	Passes	Total	Accuracy
Gemini-2.0-Flash	Python → JavaScript	164	164	100.00%
	JavaScript → Python	147	164	89.63%
GPT-3.5-Turbo-0125	Python → JavaScript	151	164	92.07%
	JavaScript → Python	148	164	90.24%

This iterative refinement process was run five times maximum as it was found that, out of the tests that were corrected from this process, 79% were corrected after one iteration and no more than four iterations benefited the pass rate. Figure 1 shows the percentage of examples fixed at each iteration of iterative refinement.

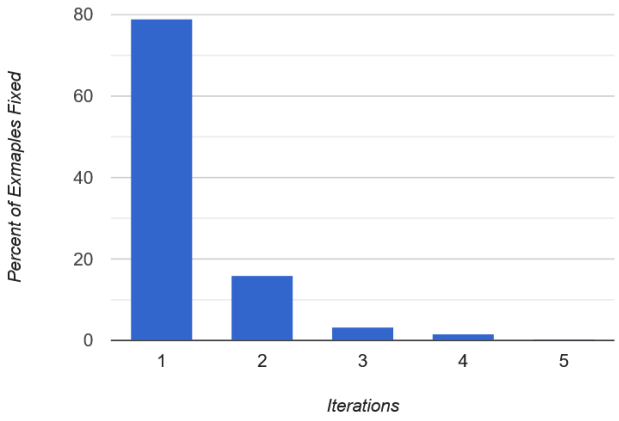

Figure 1: Iterative Refinement Iterations Pass Rate

3.4 Overall Improvement

Through all the methods previously described, the overall improvement of the model can be seen in figures 2 and 3 for the Python to JavaScript direction and JavaScript to Python direction respectively. These plots show how the number of passed tests, out of the 164 total, increases after each improvement to the prompting process. Overall, the Python to JavaScript direction performed better and the GPT-3.5 model, despite starting off with significantly worse performance, improved greatly through these methods and acheived results comparable to Gemini-2.0.

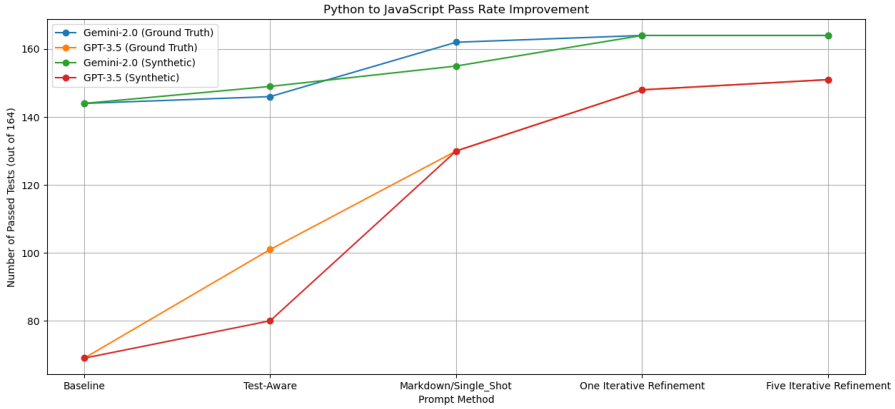

Figure 2: Python to JavaScript Pass Rate Improvement

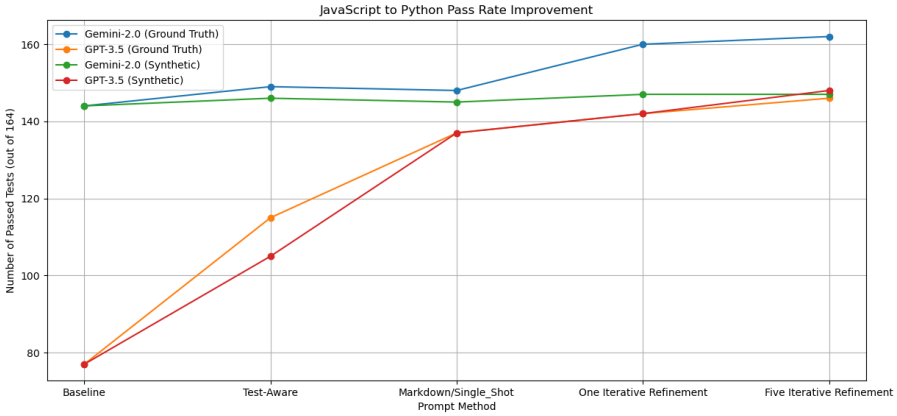

Figure 3: JavaScript to Python Pass Rate Improvement

4 Discussion

4.1 Iterative Refinement and Multiple Candidates

Both iterative refinement and multiple candidates were considered as final steps for this pipeline. A multiple candidates approach would involve querying the LLM multiple times with asking for the same code translation and designing some method to choose the best result. To choose which method would provide better results initial tests were done on an example that the pipeline failed extremely often. It was found that, when queried ten times, the multiple candidates approach was only able to generate a successful translation one time. The iterative refinement approach was able to fix the issue after only one re-query. This showed the ability to incrementally improve upon its previous attempts was beneficial and would result in a greatly reduced number of queries. Ultimately, iterative refinement was chosen as initial tests showed it vastly outperformed multiple candidates.

4.2 Language Performance Discrepancy

A notable discrepancy observed during this project was the LLM's comparatively weaker performance in understanding JavaScript versus Python. Specifically, when tasked with translating JavaScript to

Python, the model consistently underperformed relative to scenarios where Python was the source language. This disparity is likely due to the significantly larger volume of Python example code available online, which provides the LLM with richer training data. As a result, when JavaScript is used as the source, the model can struggle to fully grasp the intended program semantics, leading to inaccurate translations.

Further evidence of this limitation was noticed when analyzing results based solely on ground truth test cases. The model consistently performed better when the test cases were written in Python. This highlights a potential future improvement to the pipeline where the synthetic test generation step either directly generates python test cases or the JavaScript test cases are translated to Python before being used in the next prompt. These findings are illustrated in Table 12, where Gemini-2.0 is evaluated using different sets of ground truth test cases. The prompt used for this experiment incorporated markdown formatting and included a single example translation.

Table 12: Gemini Markdown and Single Shot Prompt (Ground Truth Tests)

Direction	Test Case Language	Passes	Total	Accuracy
Python → JavaScript	Python	162	164	98.78%
	JavaScript	155	164	94.51%
JavaScript → Python	Python	159	164	96.95%
	JavaScript	148	164	90.24%

4.3 Error Analysis

A recurring challenge in translating code between JavaScript and Python stems from the model's misinterpretation of language-specific semantics such as differences in variable scope, asynchronous behavior, and built-in function usage. While the generated code is often syntactically valid, it frequently fails to preserve the original logic. Common pitfalls include incorrect return types or formats, overly literal translations of JavaScript methods, unintended shadowing of Python built-ins, and mismatched mathematical semantics. Additionally, the absence of strict type enforcement can allow subtle bugs to pass undetected. These issues highlight the need for thorough testing and iterative refinement to ensure robust and accurate translations. It is worth noting that in many cases the direct output of the LLM would also cause errors as it would try to continue the markdown format by adding extra '“' and labelling the code language as 'Python' or 'JavaScript'. This was simple to fix as every output was cleaned so that it only contained compilable code. Some examples of common errors are as follows:

- **Incorrect Return Type or Format:** Many translations return incorrect data types or formats compared to the original code. For example:
  - JavaScript: `return '0b' + k.toString(2);`
  - Incorrect Python: `return '0b' + int(k).to_bytes(...).hex()` *(wrong format)*
- **Literal or Missing Translation of JavaScript Functions:** Functions like `.toFixed()`, `.split()`, `.reduce()`, and `.slice()` are sometimes translated too literally or not at all. For example:
  - JavaScript: `k.toString(2)`
  - Incorrect Python: `int(k).to_bytes(...)` *(incorrect translation)*
- **Shadowing Built-in Names:** JavaScript allows the use of variable names that shadow global objects like `int`, `list`, and `str`. In Python, this can cause unintended consequences. For example:
  - JavaScript: `const sumProduct = (numbers, int) => { ... };`
  - Problematic Python: `def sum_product(numbers, int):` *('int' shadows built-in)*
- **Mismatched Mathematical Semantics:** Translating mathematical expressions, especially floating-point comparisons or rounding behavior, can lead to logic errors. For instance, Python's modulo operation on floats behaves differently than JavaScript.

4.4 Threats to Validity

As seen in the results, Gemini-2.0 performed significantly better than GPT-3.5 in the baseline test. It was only through the prompt engineering and iterative refinement steps that allowed the GPT-3.5 model to perform comparably with Gemini-2.0. One contributing factor to this could be the fact that Gemini-2.0 was trained and released after the release of the Human-Eval-X dataset and is therefore not free of data leakage concerns. This is also a major contributing factor to the decision to include GPT-3.5 in this research project as it was released before Human-Eval-X.

We observed one function in JavaScript dataset is incorrectly implemented and does not meet intended behaviour, which introduced noise in the evaluation. However, this data sample is isolated and does not significantly affect the overall trends observed.

5 Conclusion

This project presents a structured pipeline for improving bi-directional code translation between Python and JavaScript using LLMs. Through systematic evaluation on the HumanEval-X dataset, we demonstrate that the inclusion of test cases, structured prompt formatting, and example code translations enhances the output reliability. Furthermore, the iterative refinement step proves effective in recovering from initial translation failures, leveraging the model's ability to self-correct with contextual feedback. Through these methods the output pass rate was able to improve from a baseline 87.80% and 44.33% to 96.65% and 91.16% (average between performance in both directions) for the Gemini-2.0 and GPT-3.5 LLMs respectively. The improvements made through prompt engineering are particularly apparent when using GPT3.5 which was released before the Human-Eval-X dataset. One key insight from our experiments is the LLM's stronger performance when Python is used as the source or when test cases are provided in Python, likely due to its richer representation in training data. These results highlight practical opportunities to improve code translation workflows, such as standardizing test case format or refining synthetic test generation. Future work could extend this approach to more language pairs and explore dynamic error detection strategies during refinement.

References

Google Cloud (2024). *Gemini 2.0 Flash Model*. `https://cloud.google.com/vertex-ai/generative-ai/docs/models/gemini/2-0-flash`. Accessed: 2025-04-12.

S. Kang, J. Yoon, and S. Yoo (2022). *Large Language Models are Few-shot Testers: Exploring LLM-based General Bug Reproduction*. Tech. rep. KAIST.

OpenAI (2023). *GPT-3.5-Turbo Model*. `https://platform.openai.com/docs/models/gpt-3.5-turbo`. Accessed: 2025-04-12.

Q. Zheng, X. Xia, X. Zou, Y. Dong, S. Wang, Y. Xue, L. Shen, Z. Wang, A. Wang, Y. Li, et al. (2023). *CodeGeeX: A Pre-trained Model for Code Generation with Multilingual Benchmarking on HumanEval-X*.

A Appendix

A.1 Example Baseline Prompt

You are given a JavaScript function below.

Your task: Translate the function to Python.
- The translated Python code should preserve all functionalities in the original JavaScript code.
- Use snake_case for all function and variable names, following Python conventions.
- Output only the translated Python code, nothing else.
{insert js code}

A.2 Example Test-Aware Prompt

You are given a JavaScript function below.

Your task: Translate the function to Python.
- The translated Python code should preserve all functionalities in the original JavaScript code.
- Use snake_case for all function and variable names, following Python conventions.
- Output only the translated Python code, nothing else.
{insert js code}

For reference, here are some test cases the original function passed, your translated function should also pass all the tests.
{insert js test case}

A.3 Example Markdown Single-Shot Prompt

You are given a JavaScript function below.
Your task: Translate the function to Python.
- The translated Python code should preserve all functionalities in the original JavaScript code.
- Use snake_case for all function and variable names, following Python conventions.
- Output only the translated Python code, nothing else.
- You are provided with an example translation for reference.
JavaScript Code Example:
```
{insert exmaple js code}
```
JavaScript Example Test Cases:
```
{inster example js test cases}
```
Python Code Example Translation:
```
{insert exmaple python code}
```
Translate the JavaScript function to Python.
JavaScript Code:
```
{insert js code}
```
JavaScript Test Cases:
```
{insert js test cases}
```
Python Code:
```
```