Multi-Programming Language Sandbox for LLMs

Shihan Dou[1*], **Jiazheng Zhang**[1*], **Jianxiang Zang**[1*], **Yunbo Tao**[1], **Weikang Zhou**[1], **Haoxiang Jia**[2], **Shichun Liu**[1], **Yuming Yang**[1], **Shenxi Wu**[1], **Zhiheng Xi**[1], **Muling Wu**[1], **Rui Zheng**[1], **Changze Lv**[1], **Limao Xiong**[3], **Shaoqing Zhang**[4], **Lin Zhang**[3], **Wenyu Zhan**[3], **Rongxiang Weng**[3], **Jingang Wang**[3], **Xunliang Cai**[3], **Yueming Wu**[5], **Ming Wen**[5], **Yixin Cao**[1], **Tao Gui**[1,6,7†], **Xipeng Qiu**[1,6,7], **Qi Zhang**[1], **Xuanjing Huang**[1†]

[1]Fudan University [2]Peking University [3]Meituan Inc
[4]Harbin Institute of Technology [5]Huazhong University of Science and Technology
[6]Shanghai Innovation Institute [7]Pengcheng Laboratory

shdou21@m.fudan.edu.cn, wengrongxiang@meituan.com, tgui@fudan.edu.cn

Abstract

We introduce MPLSandbox, an out-of-the-box multi-programming language sandbox designed to provide unified and comprehensive feedback from compiler and analysis tools for Large Language Models (LLMs). It can automatically identify the programming language of the code, compiling and executing it within an isolated sub-sandbox to ensure safety and stability. In addition, MPLSandbox integrates both traditional and LLM-based code analysis tools, providing a comprehensive analysis of generated code. It also can be effortlessly integrated into the training and deployment of LLMs to improve the quality and correctness of generated code. It also helps researchers streamline their workflows for various LLM-based code-related tasks, reducing the development cost. To validate the effectiveness of MPLSandbox, we conduct extensive experiments by integrating it into several training and deployment scenarios, and employing it to optimize workflows for a wide range of downstream code tasks. Our goal is to enhance researcher productivity on LLM-based code tasks by simplifying and automating workflows through delegation to MPLSandbox[1].

1 Introduction

Recently, researchers have become increasingly interested in the development of large language models (LLMs) for code tasks (Le et al., 2023; Shin et al., 2023). However, LLM-generated code may contain vulnerabilities and harmful programs, making it necessary to compile and execute the code within a sandbox environment (Garfinkel et al., 2003; Liang et al., 2003). Despite this necessity, most existing sandboxes focus on only one or two programming languages (Engelberth et al., 2012; Ter, 2024), and are not easily integrated into the training and deployment processes of LLMs (Cassano et al., 2022; LLM, 2024). The lack of well-developed multi-language sandbox environments significantly limits the application of LLMs in tasks involving multiple programming languages.

On the other hand, researchers commonly use various code analysis tools to enhance the quality of LLM-generated code (Liu et al., 2023; Gazzola et al., 2019). Downstream coding tasks also require these tools to seamlessly integrate with LLMs (Du et al., 2024; Lu et al., 2024). The wide variety of tools significantly increases the development difficulty and cost for researchers (Manès et al., 2019; Gentleman and Temple Lang, 2007), especially in multi-language programming scenarios. Unfortunately, there is currently no out-of-the-box code analysis toolbox that can be directly used with LLMs for various coding tasks.

To address these issues, we propose MPLSandbox, an out-of-the-box multi-programming language sandbox designed to provide unified compiler feedback for LLM-generated code. It also integrates over 40 code analysis tools to deliver comprehensive analysis results from various perspectives. MPLSandbox can be seamlessly integrated into the training and deployment of LLMs, enhancing their performance on various code tasks and significantly streamlining users' workflows. MPLSandbox consists of three core modules: (1) the "Multi-Programming Language Sandbox Environment", which compiles and executes code to

[*] Equal contribution.
[†] Corresponding author.
[1]MPLSandbox has been used for large-scale training and various downstream code tasks such as code data distillation and code optimization at Meituan Inc. The installable package is available at: https://github.com/Ablustrund/MPLSandbox. The demonstration Video is available at: https://youtu.be/ecpspPrkYrQ. MPLSandbox is licensed under the Apache 2.0 open-source license.

Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 3: System Demonstrations), pages 40–50

provide unified compiler feedback; (2) the "Code Analysis Module", which includes various analysis tools to offer comprehensive analysis; and (3) the "Information Integration Module", which integrates compilation feedback and analysis results to accomplish a range of complex code-related tasks.

For the first module, the code and unit tests are sent to the sub-sandbox of the corresponding programming language for isolated execution. The sandbox ensures the program executes safely without jeopardizing the external environment or interrupting the training process. The second module provides a comprehensive code analysis from various perspectives, such as static analysis (*e.g.,* potential bug detection and code smell analysis) and dynamic analysis (*e.g.,* fuzz testing and efficiency analysis). This module can also analyze other key aspects beyond the code, such as evaluating unit test coverage to help researchers improve the quality of these unit tests. Finally, the third module integrates these results to improve the quality of generated code and helps users enhance the convenience of applying LLMs in various downstream tasks. Specifically, the features of our proposed MPLSandbox include:

- **Security and stability.** Sub-sandboxes ensure that programs are compiled and executed in isolation from the training environment. This prevents LLM-generated code containing malicious vulnerabilities or bugs from harming the external environment. Moreover, various integrated vulnerability and bug detection tools further ensure safety.

- **Multi-programming language support.** We are the first to propose a multi-programming language sandbox that integrates over 40 code analysis tools. MPLSandbox can automatically identify the programming language of the code, assign it to the corresponding sub-sandbox, and thoroughly analyze it using various tools. This significantly reduces the development cost for researchers in deploying and developing LLMs for downstream code tasks.

- **Usability and extensibility.** MPLSandbox integrates various analysis tools for each programming language, and users can also effortlessly design tool templates to integrate their tools into the sandbox. Moreover, users can easily construct prompt templates to combine compiler feedback and analysis results to accomplish code tasks.

- **Distributed architecture.** MPLSandbox is designed for distributed deployment. In large-scale training scenarios, training nodes can access any MPLSandbox nodes. This setup offers greater efficiency compared to deployments where both training nodes and sandbox nodes are co-located on a single machine.

We conduct extensive experiments on three application scenarios to validate MPLSandbox: verifying code at inference time, providing compiler feedback in reinforcement learning, and self-correcting and optimizing code. Moreover, we showcase that it can streamline workflows for diverse code tasks like unit test generation, vulnerability localization, and code translation. Results demonstrate that MPLSandbox integrates easily into all scenarios, reducing development costs.

MPLSandbox is the first multi-programming language sandbox with over 40 analysis tools, simplifying the use of LLMs in code tasks. Its ease of use and flexible module combination make it effective for many downstream tasks, while keeping development costs low for researchers. We hope our tool drives further research in this area.

2 MPLSandbox

In this section, we introduce the architecture, pipeline, and usage of MPLSandbox.

2.1 Architecture

Our tool is an out-of-the-box multi-programming language sandbox designed to provide unified compiler feedback and comprehensive code analysis, enabling researchers to thoroughly analyze LLM-generated code in any programming language while significantly reducing development costs. It also can streamline LLMs' training and deployment workflows for various code tasks. The architecture of MPLSandbox is shown in Figure 1. If no programming language type is specified, the built-in rule-based and model-based parsers automatically detect the code's language. Our tests on 10 million lines of code show that the classification error rate is less than 0.1%. Subsequently, the code is comprehensively analyzed by three core modules: (1) Multi-Programming Language Sandbox Environment, (2) Code Analysis Module, and (3) Information Integration Module.

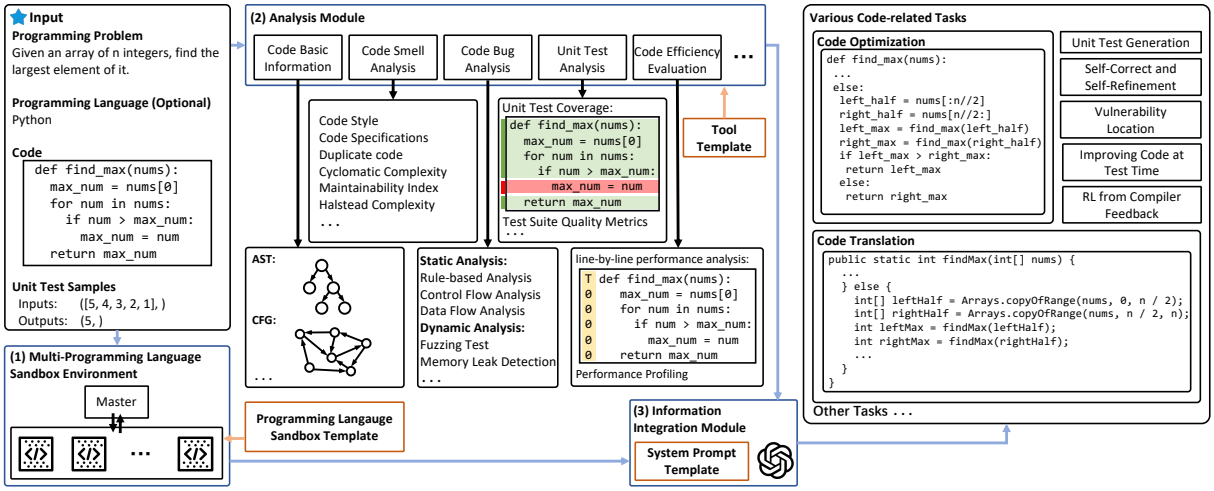

Figure 1: The architecture of MPLSandbox. It comprises three core modules: (1) Multi-Programming Language Sandbox Environment, (2) Code Analysis Module, and (3) Information Integration Module. The Multi-Programming Language Sandbox Environment can provide unified compiler feedback by compiling and executing the code. The Code Analysis Module contains multiple traditional analysis tools to offer a comprehensive analysis report from numerous perspectives. The Information Integration Module integrates compilation feedback and various analysis results to accomplish a range of complex code-related tasks.

Multi-Programming Language Sandbox Environment. Based on the specified programming language, the module first sends the code and unit test samples into the corresponding sub-sandbox for secure compilation and execution. The sub-sandbox is a container isolated from the main environment to prevent potential vulnerabilities in the code from affecting the external environment. It is configured with resource constraints, such as maximum memory limit, execution time, and PIDs limit, to prevent resource overuse that could crash the sandbox. To further ensure stability during LLM training and deployment, a driver node continuously monitors the sandbox node in real-time and can automatically restart it in case of a crash due to unknown reasons. It also analyzes runtime and resource usage, and reports analysis results during both program execution and the execution of analysis tools (detailed in the Code Analysis Module).

Each programming language sub-sandbox comes pre-installed with widely used dependency libraries. Users can also write a configuration file to easily install additional libraries. It can report missing libraries based on compiler feedback, allowing users to identify and install required dependencies effortlessly. We have predefined eight commonly used programming languages: Python, Java, C++ (C), C#, Bash, Go, JavaScript (JS), and TypeScript (TS). Expanding to additional programming languages is straightforward. Users can create their own sub-sandbox and seamlessly integrate it into the sandbox environment.

Code Analysis Module integrates over 40 various analysis tools to provide a comprehensive report on the code from various perspectives. It can also assess key aspects beyond the code, such as evaluating unit test coverage to help researchers improve the quality of their unit test samples. We categorized these analysis tools into five groups based on their purpose and analysis results: (1) basic information analysis, (2) code smell analysis, (3) code vulnerability analysis, (4) unit test analysis, and (5) code efficiency evaluation.

(1) Basic information analysis provides detailed information on code structure and semantics, such as Abstract Syntax Trees (AST) and Control Flow Graphs (CFG), to help LLMs and users better understand the code. This information can enhance LLM performance in tasks like code completion, refactoring, security analysis, and code translation (Zhou et al., 2025; Wan et al., 2024; Liu et al., 2025). **(2) Code smell analysis** identifies patterns in code that may indicate issues affecting maintainability, readability, and extensibility, such as code complexity, overengineering, and duplicated code. It can significantly assist in various tasks, including improving code quality, aiding in code reviews by identifying potential issues, offering refactoring suggestions for cleaner code, and enhancing code understanding through contextual and structural insights. **(3) Code bug analysis** is essential in software development for ensuring quality and sta-

Type	Python	Java	C++ (C)	C#	Bash	Go	JavaScript	TypeScript
Basic Information Analysis	ASTPretty & Pyflowchart	Javalang & Soot	Clang	Roslyn	-	GoAst Viewer & Angr	Joern	Ts-morph
Code Smell Analysis	Pylint & Radon	Pmd	CPPCheck	StyleCop.Analyzers	ShellCheck	golangci-lint	ESLint & Shkjem	ESLint & TSLint
Code Bug Analysis	Bandit	Checkstyle	PVS-Studio & CPPCheck	SonarQube	Shellcheck	govulncheck & gosec	NodeJsScan	Snyk
Unit Test Analysis	Coverage	Jacoco	GCOV	Coverlet	shcov	gocov	Istanbul	Istanbul
Code Efficiency Evaluation	Line_profile	Jprofile	Benchmark.NET	BenchmarkDotNet	bashprof	pprof	V8 Profiler	V8 Profiler

Table 1: Overview of code analysis tools integrated within MPLSandbox.

bility, comprising both static and dynamic analysis. The former detects errors and vulnerabilities without executing code, while the latter identifies runtime issues, including through fuzz testing. These tools assist in various aspects, such as improving code security, aiding LLM self-debugging and self-correction, and generating comprehensive documentation, making the code more reliable. **(4) Unit test analysis** involves evaluating the effectiveness and coverage of unit tests to ensure code quality and reliability. It helps LLMs identify uncovered code lines, generate new test cases, diagnose errors, and offer code quality suggestions, making development and testing more efficient and automated. **(5) Code efficiency evaluation** assesses code performance and resource utilization by analyzing aspects such as time and space complexity, line-level execution time, and resource usage. It can enhance LLM performance in various code tasks by identifying inefficiencies, pinpointing bottlenecks, providing optimization suggestions, enabling automated improvements, and offering continuous feedback.

Table 1 lists over 40 commonly used tools integrated for each programming language. Users can also easily add their analysis tools by writing tool templates. These tools provide comprehensive information about the code, helping LLMs and users better understand and optimize code. Moreover, the combination of these tools with LLMs enhances their performance in various code tasks. We demonstrate the ease of use and applicability of MPLSandbox in several tasks, as detailed in Section 3 and 3.3.

Information Integration Module collects compiler feedback from the Multi-Programming Language Sandbox Environment and various analysis results from the Code Analysis Module to enhance the quality of generated code and help LLMs accomplish complex code-related tasks. It includes rich templates to reconstruct these results and then feed them into LLMs. Users can also create custom prompt templates to combine these results, streamlining LLM workflows in various downstream tasks and reducing development costs. For example, users can enable LLMs to generate diverse and comprehensive unit tests based on unit test analysis and compiler feedback, and improve code translation by leveraging structural, semantic, and execution information. More usage cases are provided in Appendix C and our GitHub repository.

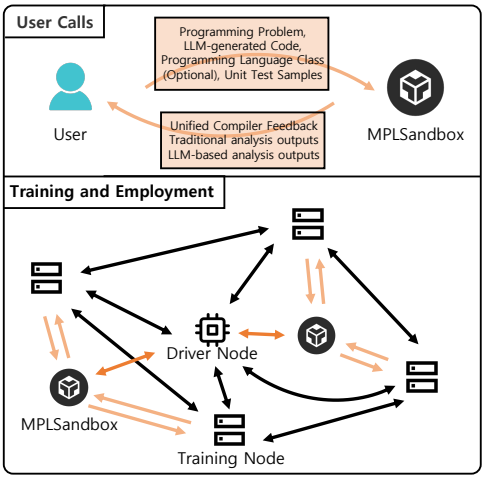

Figure 2: The pipeline of MPLSandbox. It can be deployed as either a standalone system for individual users, or as a distributed system for large-scale LLM training and deployment.

2.2 Pipeline

MPLSandbox can be deployed as a standalone system for individual users or several LLMs, or as a distributed system for large-scale training and deployment scenarios. Figure 2 shows its pipeline in these two scenarios. First, users can deploy MPLSandbox on their personal computers or remote servers and easily invoke it via an IP address and port number for comprehensive analysis and evaluation of LLM-generated code. Users can also integrate MPLSandbox into small-scale LLM training and deployment workflows to enhance the effectiveness of LLMs, such as deploying it to verify the codes at inference time or to provide compiler feedback in Reinforcement Learning from Compiler Feedback (RLCF). More various usage cases provided in Appendix C and our GitHub repository

show that MPLSandbox can optimize workflows for various downstream code tasks.

Moreover, MPLSandbox can be integrated into large-scale distributed training and deployment environments. We can deploy multiple sandbox node servers and manage them centrally through a driver node. Sandbox nodes can be custom-assigned to training nodes to provide services, and to prevent memory and CPU pressure, sandbox nodes and training nodes can be deployed separately. MPLSandbox streamlines the workflow of large-scale LLM training and deployment, effectively saving researchers' development time.

2.3 Usage

MPLSandbox is designed with flexibility in mind, allowing users to configure workflows and integrate their analysis tools, while providing appropriate abstractions to mitigate concerns about low-level implementation details. It is ready-to-use and can be easily invoked with just a few lines of code. We briefly outline the MPLSandbox's analysis process for a code segment through a case. The case can be represented as follows:

```
question = '''
-----Description-----
Write a example function calculation()
    ...'''
code = '''def calculation(): ...'''
unit_cases =
{"inputs":["51","120","211"],
 "required_outputs":['[1, 3, ... 1326]',
                     '[1, 2, ... 1728001]',
                     '[1, 2,... 9129330]']}
lang = "AUTO" # Automatic language detection
case = {"code": code, "question": question,
"unit_cases": unit_cases, "lang": lang}
```

It contains the code segment to be verified and other information for compiling, executing, and analyzing this code including the description, unit tests, and the optional programming language type. The language type also can be automatically detected.

We first instantiate a verification class by using its dictionary or JSON file. Then, we can simply obtain the analysis results of this code by invoking the `run` method:

```
from MPLSandbox import MPLSANDBOX
tobeverified = MPLSANDBOX(case)
report = tobeverified.run(analysis="all")
# support selecting specific analysis
```

The executor first calls the Code Analysis Module to analyze the code from five different perspectives. It then integrates these analysis results through the Information Integration Module and returns the final results to the user. Users can easily specify the code analysis information they wish to obtain through the analysis parameter. More detailed usage methods and cases are provided in Appendix C and our GitHub repository.

3 Applications

In this section, we showcase three main application scenarios of MPLSandbox in improving the quality of LLM-generated code and helping users streamline LLM workflows of various downstream code tasks. We also provide more application scenarios and cases in a wide range of tasks in Section 3.3 and our GitHub repository.

3.1 Setup

We conduct all experiments using the **TACO** dataset (Li et al., 2023), which comprises programming problems sourced from the **APPS+** (Dou et al., 2024) dataset, the **CodeContests** dataset (Li et al., 2022), and various contest sites. We validate our tool on a wide range of LLMs, including DeepSeek-Coder-Instruct-6.7B (Guo et al., 2024), DeepSeek-Coder-V2-Lite-Instruct-16B (Zhu et al., 2024), Qwen2.5-Coder-1.5B-Instruct (Team, 2024), Qwen2.5-Coder-7B-Instruct (Team, 2024), Codestral-v0.1-22B (mis, 2024), Llama-3.1-Instruct-70B (Dubey et al., 2024), and GPT-4o (OpenAI, 2023), to enhance their ability on code tasks. We report Pass@k results (Chen et al., 2021) in our experiments. For Pass@1 and Pass@10 settings, the sampling temperatures are set to 0.2 and 0.8, respectively. All inference experiments are conducted on a single node equipped with eight A100-80G GPUs, while all training experiments are conducted on 16 training nodes and two MPLSandbox nodes. Detailed system templates for multi-programming language code generation and other code tasks, descriptions of the foundation models, and implementation information, including RL training specifics, are provided in our GitHub.

3.2 Results

As a Verifier at inference time. First, We integrate MPLSandbox into the deployment environment of LLMs to verify the correctness of generated code at inference time, as shown in Table 2. Results show that it reliably verifies model-generated code in multiple programming languages. This can simplify deployment scenarios such as code evaluation, data production, filtering, and automated testing.

Model	Size	Pass@K	Python	Java	C++(C)	C#	Go	Bash	JavaScript	TypeScript
Qwen2.5-Coder-Instruct	1.5B	K=1	2.4%	2.8%	2.8%	0.4%	1.1%	0.0%	0.4%	0.4%
		K=10	13.9%	4.9%	8.5%	7.3%	4.9%	4.5%	2.4%	1.7%
Qwen2.5-Coder-Instruct	7B	K=1	7.0%	14.3%	11.9%	11.5%	3.5%	4.9%	9.1%	3.8%
		K=10	24.7%	23.7%	32.1%	28.6%	23.7%	20.6%	25.4%	17.8%
DeepSeek-Coder-Instruct	6.7B	K=1	9.4%	10.5%	9.1%	8.0%	3.8%	2.4%	7.0%	3.1%
		K=10	23.7%	24.7%	22.3%	25.1%	21.6%	16.4%	21.6%	15.3%
DeepSeek-Coder-V2-Lite-Instruct	16B	K=1	29.6%	26.8%	25.1%	23.7%	10.5%	5.6%	12.9%	8.0%
		K=10	50.2%	47.7%	44.6%	42.9%	35.5%	19.9%	39.4%	25.1%
Codestral-v0.1	22B	K=1	9.8%	21.3%	22.0%	20.2%	12.2%	10.1%	9.8%	7.0%
		K=10	34.2%	41.8%	38.7%	41.1%	34.8%	28.9%	34.8%	28.6%
Llama-3.1-Instruct	70B	K=1	15.0%	17.4%	15.7%	13.2%	6.6%	7.4%	9.4%	6.1%
		K=10	38.0%	38.3%	34.5%	35.5%	35.5%	17.1%	33.5%	14.6%
GPT-4o	-	K=1	39.3%	47.4%	46.3%	16.0%	43.6%	33.8%	44.6%	40.4%
		K=10	52.6%	68.6%	65.9%	47.4%	64.5%	58.2%	66.2%	63.4%

Table 2: Results of integrating MPLSandbox into the deployment environment. It indicates that it provides reliable verification and feedback.

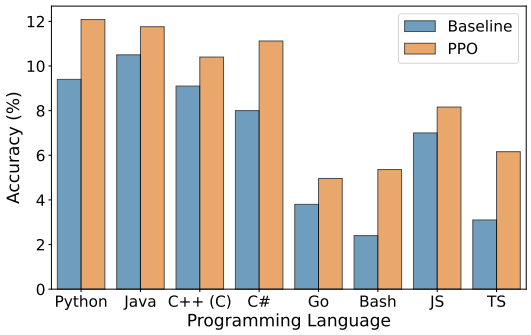

Figure 3: Pass@1 results on improvement in reinforcement learning from compiler feedback. Users can effortlessly obtain reliable compiler feedback and streamline their LLM training workflow through MPLSandbox.

For instance, it is used by the data engineering and evaluation teams of Meituan Inc. to bulk filter LLM-generated code and provide compiler feedback in various evaluation environments.

Providing feedback signals in RL. We validate its effectiveness in providing compiler feedback by integrating it into RLCF to enhance LLM code generation. We initialize the policy model using DeepSeek-Coder-Instruct and employ PPO as the RL algorithm. The optimization objectives and reward design are detailed in our GitHub repository. Experimental results, shown in Figure 3, indicate significant improvements in LLM code generation, demonstrating the stability and accuracy of our tool's feedback. It enables users to bypass trivial tasks like isolating and building multi-language execution environments. By simply invoking MPLSandbox, users can focus more on developing and optimizing their training algorithms.

We also provide more application scenarios and cases in Appendix C, such as unit test generation, vulnerability localization, and code translation. These indicate the effectiveness of MPLSandbox for various workflows which can significantly reduce development effort.

Self-correction and self-optimization. Self-correcting and optimizing LLM-generated code is essential yet often complex and laborious, necessitating detailed information about code errors, complexity, execution efficiency, and adherence to coding standards, which in turn requires numerous cumbersome code analysis tools. With MPLSandbox, users can seamlessly analyze LLM-generated code and achieve self-debugging and self-refinement. To demonstrate its utility, we employed our tool to enable GPT-4 to both correct erroneous code and refine accurate code. System prompts for these operations are available in our GitHub repository. After one round of self-correction, Pass@1 results improve by 3.7% for Python, 4.9% for Java, 2.7% for C++ (C), 6.5% for C#, 5.0% for Go, 4.8% for Bash, 4.1% for JavaScript, and 3.1% for TypeScript. These results indicate that it can provide accurate compiler feedback across various programming languages, enabling GPT-4 to solve more programming problems. Moreover, the code produced exhibits greater compliance with programming specifications, as detailed on our GitHub repository.

Case study on self-optimization. We utilize an instance from the test set to illustrate the process of self-refinement, as shown in Figure 4. It begins with Code Smell Analysis for smell detection, identifying code issues such as No Docstrings and Lack of Comments, Unclear Variable Naming, Hard-coded Limits, High Complexity, and Redundant Sorting. Subsequently, the built-in LLM-based system proposes corresponding improvements for these suggestions. Finally, these suggestions are incorporated into system prompts to achieve self-

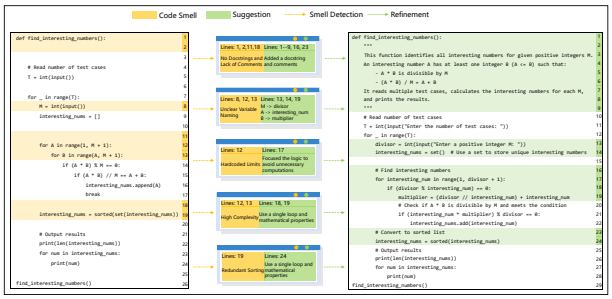

Figure 4: Self-refinement process.

refinement. The supplementary material in our GitHub repository also shows the results of analyzing certain metrics of the code before and after refinement using maintainability analysis, quantifying the effectiveness of the refinement. Results show that the overall cyclomatic complexity and Halstead Volume of the code have decreased, resulting in an increase in the Maintainability Index, further showing the positive feedback of the entire refinement on code maintainability.

3.3 Examples for Application Scenarios

We also showcase how to use MPLSandbox for other tasks, including unit test generation, code translation, and vulnerability localization, significantly improving development efficiency.

Unit test generation. When code is more complicated, unit tests often struggle to comprehensively cover the generated code, leaving untested code segments at risk of latent defects. Prior work (Jiang et al., 2024) shows that users can identify uncovered code segments by using unit test analysis tools and integrate them into prompts to drive LLMs to generate supplementary test cases to validate uncovered segments. MPLSandbox streamlines this process, allowing users to accomplish this task by directly designing system prompts, thereby enhancing the performance of test completeness and reliability.

Code translation. LLMs have been extensively applied in code translation. Research ((Tao et al., 2024; Luo et al., 2024)) shows that integrating information such as unit tests and CFGs into system prompts can significantly enhance LLMs' code comprehension capabilities, improving translation success rates. MPLSandbox can effortlessly accomplish code translation tasks by integrating the above helpful results from the code analysis module into the information integration module using system prompts.

Vulnerability location. LLMs also empower developers to identify code security vulnerabilities. Some work ((Lu et al., 2024; Akuthota et al., 2023)) shows that integrating results from static vulnerability analysis tools into prompts enhances detection accuracy, enabling function-level vulnerability localization. MPLSandbox enables users to achieve this task by directly utilizing the required analysis results and constructing their system prompts, significantly reducing development costs.

The system prompts used in all scenarios are provided in our GitHub repository.

4 Conclusion

We introduce MPLSandbox, an out-of-the-box multi-programming language sandbox for unified compiler feedback and comprehensive code analysis of LLM-generated code. Researchers can use it to analyze codes and integrate it into training and deployment to improve code correctness and quality. MPLSandbox can also enhance LLM performance on various code tasks through flexible tool combinations. Our goal is to support and advance further research in LLMs for software engineering by simplifying the complexity of training and employing LLMs in various code tasks.

Limitations

First, although we have pre-installed numerous dependency packages for each programming language sub-sandbox, it is evident that we cannot install every package a user needs. However, users can easily install the required packages by using scripts. Secondly, we have built-in support for eight commonly used programming languages. Users can simply create sub-sandboxes to support additional programming languages. In the future, we plan to support more programming languages. Finally, our sandbox requires Docker to run. If the user's training node is itself a Docker container, this sandbox cannot run within it, as the Docker cannot be nested inside another Docker container. To resolve this, we can run the sandbox in a distributed manner on a physical machine and remotely invoke the sandbox via IP address and port number.

Acknowledgement

The authors wish to thank the anonymous reviewers for their helpful comments. This work was partially funded by the Major Key Project of PCL under Grant PCL2024A06, National Natural Science Foundation of China (No.

62476061,62206057,62076069), Shanghai Rising-Star Program (23QA1400200), Natural Science Foundation of Shanghai (23ZR1403500), Program of Shanghai Academic Research Leader under grant 22XD1401100.

References

2024. Dify-sandbox. https://github.com/langgenius/dify-sandbox.

2024. Llmsandbox. https://hackernoon.com/introducing-llm-sandbox-securely-execute-llm-generated-code-with-ease.

2024. mistralai.

2024. Promptfoo. https://www.promptfoo.dev/docs/guides/sandboxed-code-evals/.

2024. Terrarium. https://github.com/cohere-ai/cohere-terrarium.

Vishwanath Akuthota, Raghunandan Kasula, Sabiha Tasnim Sumona, Masud Mohiuddin, Md Tanzim Reza, and Md Mizanur Rahman. 2023. Vulnerability detection and monitoring using llm. In *2023 IEEE 9th International Women in Engineering (WIE) Conference on Electrical and Computer Engineering (WIECON-ECE)*, pages 309–314. IEEE.

Federico Cassano, John Gouwar, Daniel Nguyen, Sydney Nguyen, Luna Phipps-Costin, Donald Pinckney, Ming-Ho Yee, Yangtian Zi, Carolyn Jane Anderson, Molly Q Feldman, et al. 2022. Multipl-e: A scalable and extensible approach to benchmarking neural code generation. *arXiv preprint arXiv:2208.08227*.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Shihan Dou, Yan Liu, Haoxiang Jia, Limao Xiong, Enyu Zhou, Junjie Shan, Caishuang Huang, Wei Shen, Xiaoran Fan, Zhiheng Xi, et al. 2024. Stepcoder: Improve code generation with reinforcement learning from compiler feedback. *arXiv preprint arXiv:2402.01391*.

Xiaohu Du, Ming Wen, Jiahao Zhu, Zifan Xie, Bin Ji, Huijun Liu, Xuanhua Shi, and Hai Jin. 2024. Generalization-enhanced code vulnerability detection via multi-task instruction fine-tuning. *arXiv preprint arXiv:2406.03718*.

Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. 2024. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*.

Markus Engelberth, Jan Göbel, Christian Schönbein, and Felix C Freiling. 2012. Pybox-a python sandbox.

Tal Garfinkel, Mendel Rosenblum, et al. 2003. A virtual machine introspection based architecture for intrusion detection. In *Ndss*, volume 3, pages 191–206. San Diega, CA.

L. Gazzola, D. Micucci, and L. Mariani. 2019. Automatic software repair: A survey. *IEEE Transactions on Software Engineering*, 45(01):34–67.

Robert Gentleman and Duncan Temple Lang. 2007. Statistical analyses and reproducible research. *Journal of Computational and Graphical Statistics*, 16(1):1–23.

Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y. Wu, Y. K. Li, Fuli Luo, Yingfei Xiong, and Wenfeng Liang. 2024. Deepseek-coder: When the large language model meets programming – the rise of code intelligence.

Zongze Jiang, Ming Wen, Jialun Cao, Xuanhua Shi, and Hai Jin. 2024. Towards understanding the effectiveness of large language models on directed test input generation. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, pages 1408–1420.

Hung Le, Hailin Chen, Amrita Saha, Akash Gokul, Doyen Sahoo, and Shafiq Joty. 2023. Codechain: Towards modular code generation through chain of self-revisions with representative sub-modules. In *The Twelfth International Conference on Learning Representations*.

Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven Chu Hong Hoi. 2022. Coderl: Mastering code generation through pretrained models and deep reinforcement learning. *Advances in Neural Information Processing Systems*, 35:21314–21328.

Rongao Li, Jie Fu, Bo-Wen Zhang, Tao Huang, Zhihong Sun, Chen Lyu, Guang Liu, Zhi Jin, and Ge Li. 2023. Taco: Topics in algorithmic code generation dataset. *arXiv preprint arXiv:2312.14852*.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. 2022. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097.

Zhenkai Liang, VN Venkatakrishnan, and R Sekar. 2003. Isolated program execution: An application transparent approach for executing untrusted programs. In *19th Annual Computer Security Applications Conference, 2003. Proceedings.*, pages 182–191. IEEE.

Jianing Liu, Guanjun Lin, Huan Mei, Fan Yang, and Yonghang Tai. 2025. Enhancing vulnerability detection efficiency: An exploration of light-weight llms with hybrid code features. *Journal of Information Security and Applications*, 88:103925.

Jiate Liu, Yiqin Zhu, Kaiwen Xiao, Qiang Fu, Xiao Han, Wei Yang, and Deheng Ye. 2023. Rltf: Reinforcement learning from unit test feedback. *arXiv preprint arXiv:2307.04349*.

Siyao Liu, He Zhu, Jerry Liu, Shulin Xin, Aoyan Li, Rui Long, Li Chen, Jack Yang, Jinxiang Xia, ZY Peng, et al. 2024. Fullstack bench: Evaluating llms as full stack coder. *arXiv preprint arXiv:2412.00535*.

Guilong Lu, Xiaolin Ju, Xiang Chen, Wenlong Pei, and Zhilong Cai. 2024. Grace: Empowering llm-based software vulnerability detection with graph structure and in-context learning. *Journal of Systems and Software*, 212:112031.

Yang Luo, Richard Yu, Fajun Zhang, Ling Liang, and Yongqiang Xiong. 2024. Bridging gaps in llm code translation: Reducing errors with call graphs and bridged debuggers. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, pages 2448–2449.

Valentin JM Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J Schwartz, and Maverick Woo. 2019. The art, science, and engineering of fuzzing: A survey. *IEEE Transactions on Software Engineering*, 47(11):2312–2331.

OpenAI. 2023. Gpt-4 technical report. *Preprint*, arXiv:2303.08774.

Houxing Ren, Mingjie Zhan, Zhongyuan Wu, Aojun Zhou, Junting Pan, and Hongsheng Li. 2024. Reflectioncoder: Learning from reflection sequence for enhanced one-off code generation. *arXiv preprint arXiv:2405.17057*.

Jiho Shin, Clark Tang, Tahmineh Mohati, Maleknaz Nayebi, Song Wang, and Hadi Hemmati. 2023. Prompt engineering or fine tuning: An empirical assessment of large language models in automated software engineering tasks. *arXiv preprint arXiv:2310.10508*.

Parshin Shojaee, Aneesh Jain, Sindhu Tipirneni, and Chandan K Reddy. 2023. Execution-based code generation using deep reinforcement learning. *arXiv preprint arXiv:2301.13816*.

André Silva, João F Ferreira, He Ye, and Martin Monperrus. 2023. Mufin: Improving neural repair models with back-translation. *arXiv preprint arXiv:2304.02301*.

Qingxiao Tao, Tingrui Yu, Xiaodong Gu, and Beijun Shen. 2024. Unraveling the potential of large language models in code translation: How far are we? *arXiv preprint arXiv:2410.09812*.

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

Yao Wan, Zhangqian Bi, Yang He, Jianguo Zhang, Hongyu Zhang, Yulei Sui, Guandong Xu, Hai Jin, and Philip Yu. 2024. Deep learning for code intelligence: Survey, benchmark and toolkit. *ACM Computing Surveys*, 56(12):1–41.

Tianyang Zhou, Haowen Lin, Somesh Jha, Mihai Christodorescu, Kirill Levchenko, and Varun Chandrasekaran. 2025. Llm-driven multi-step translation from c to rust using static analysis. *arXiv preprint arXiv:2503.12511*.

Qihao Zhu, Daya Guo, Zhihong Shao, Dejian Yang, Peiyi Wang, Runxin Xu, Y Wu, Yukun Li, Huazuo Gao, Shirong Ma, et al. 2024. Deepseek-coder-v2: Breaking the barrier of closed-source models in code intelligence. *arXiv preprint arXiv:2406.11931*.

A Related Work

LLMs are increasingly popular in software engineering applications (Ren et al., 2024; Le et al., 2022). However, the code generated by these models can contain malicious vulnerabilities. To ensure security and stability, and provide robust monitoring capabilities, it is essential to execute these compilation and execution processes within an isolated sandbox environment (Garfinkel et al., 2003; Liang et al., 2003). Despite this necessity, the development of open-source sandboxes is still in its infancy. Most sandboxes developed for LLM-generated code are typically focused on a single or two programming languages (Engelberth et al., 2012; pro, 2024; Dif, 2024). MultiPL-E (Cassano et al., 2022), LLMSandbox (LLM, 2024), and SandboxFusion (Liu et al., 2024) are multi-programming language sandboxes. However, MultiPL-E is limited to its MultiPL-E dataset, which is hard to integrate with online training tasks. LLMSandbox uses standard images for its environment, which lacks numerous commonly used dependency libraries. MPLSandbox was released prior to SandboxFusion. Moreover, our tool integrates over 40 diverse code analysis tools, providing comprehensive feedback signals such as static analysis and efficiency evaluation. Moreover, applying LLMs to code tasks is often accompanied by the use of a plethora of code analysis tools (Shojaee et al., 2023; Silva et al., 2023). Researchers usually spend significant time and effort on tasks like environment setup and resolving versioning and dependency issues.

B Docker Containerization Overhead Analysis

As shown in Figure 5, the introduction of Docker containerization in MPLSandbox incurs measurable but justifiable resource overhead: CPU utilization increases by 1-5%, and memory consumption rises by 7-80MB across programming languages. This overhead primarily stems from virtualization penalties in process scheduling and memory management. For instance, C++ exhibits the highest CPU impact (+5%) due to compilation-intensive operations, while Java shows the most significant memory increase (+80MB) due to JVM optimization constraints within containers. Crucially, this trade-off delivers essential security and stability benefits: Docker effectively isolates malicious code execution, prevents system-wide failures through resource constraints, and ensures consistent environment reproducibility. Through architectural optimizations including warm sandbox pools that reduce startup latency by 90% and distributed scheduling that isolates training resources, MPLSandbox effectively contains this overhead to under 5% of total processing time in production deployments, validating the design choice as a net positive for secure, reliable code analysis across diverse programming environments.

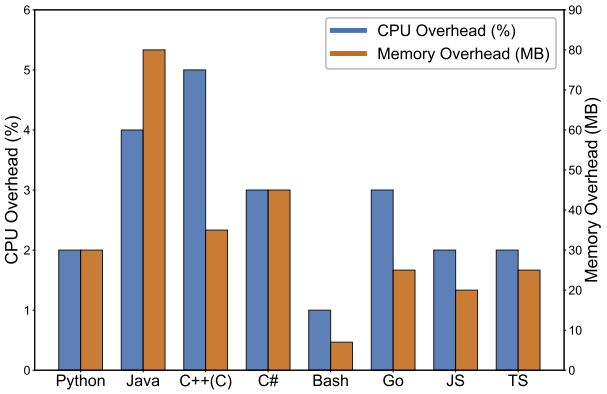

Figure 5: Docker containerization overhead analysis in MPLSandbox.

C Case Study on Usage

In this section, we conduct case studies centered around the five analysis methods based on the aforementioned configuration example in Section 2.3.

Code Basic Analysis returns a Basic Feedback along with Abstract Syntax Tree (AST) and Control Flow Graph (CFG). As shown in Figure 6, the basic feedback includes fields such as Reward, Compiler Feedback, Correct Rate, Unit Inputs, Required Outputs and Language. From the compiler feedback, it can be seen that the code has successfully passed all unit tests, achieving a correct rate of 1.0 and a reward of 1.0.

The AST presents the syntactic structure of the code in a tree diagram, where each node represents a syntactic element in the code. This structure helps to understand the logic and hierarchical relationships of the code, facilitating code optimization and error detection. The CFG graphically displays the execution paths and decision points of the code, including basic blocks and edges, which helps to reveal the execution order of the program and potential branching conditions.

Code Smell Analysis and **Code Bug Analysis** modules are designed to identify potential issues or vulnerabilities in the code, reporting specific line numbers along with the categories of smells or bugs. To better demonstrate this functionality,

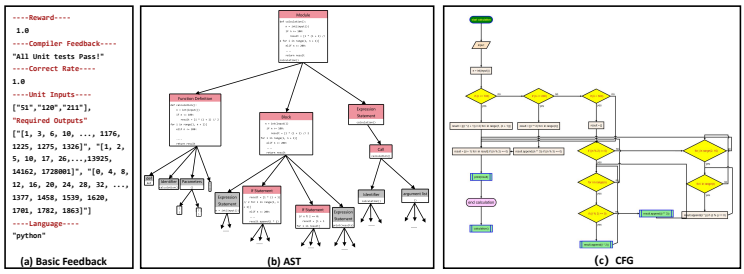

Figure 6: Reports of code basic analysis.

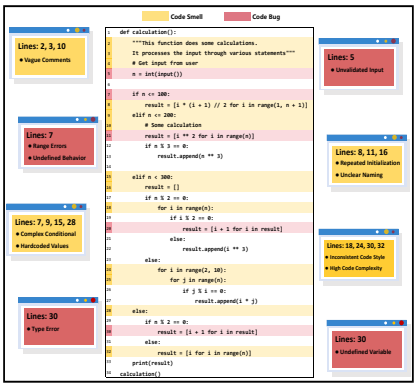

Figure 7: Reports of code smell and bug analysis.

Unit Input: "51"				Unit Input: "120"				Unit Input: "211"				Code Lines	
Hits	Time	Per Hits	%Time	Hits	Time	Per Hits	%Time	Hits	Time	Per Hits	%Time		
												def calculation():	1
1	2.3	2.3	7	1	2.4	2.4	2.2	1	2.5	2.5	0.7	n = int(input())	2
1	0.7	0.7	2	1	0.7	0.7	0.6	1	0.7	0.7	0.2	if n <= 100:	3
1	17.1	17.1	52									result = [i * (i + 1) // 2 for i in range(1, n + 1)]	4
				1	0.4	0.4	0.4	1	0.4	0.4	0.1	elif n <= 200:	5
				1	58.1	58.1	54.7					result = [i ** 2 for i in range(n)]	6
				1	0.8	0.8	0.8					if n % 3 == 0:	7
				1	1.7	1.7	1.6					result.append(n ** 3)	8
								1	0.4	0.4	0.1	elif n < 300:	9
								1	0.4	0.4	0.1	result = []	10
								1	0.8	0.8	0.1	if n % 2 == 0:	11
												for i in range(n):	12
												if i % 2 == 0:	13
												result.append(i * 2)	14
												else:	15
												result.append(i ** 3)	16
												else:	17
								9	4.6	0.5	0.3	for j in range(2, 10):	18
								1696	543.7	0.3	39.4	for i in range(n):	19
								1688	593.7	0.4	43.0	if i % j == 0:	20
								391	189.0	0.5	13.7	result.append(i * j)	21
1	0.9	0.9	2.7	1	0.6	0.6	0.6	1	0.6	0.6	0	if n % 2 == 0:	22
				1	23.2	23.2	21.8					result = [i + 1 for i in result]	23
1	12	12	36.4	1	18.3	18.3	17.2	1	27	27	7.7	return result	24

Figure 8: Reports of code efficiency evaluation.

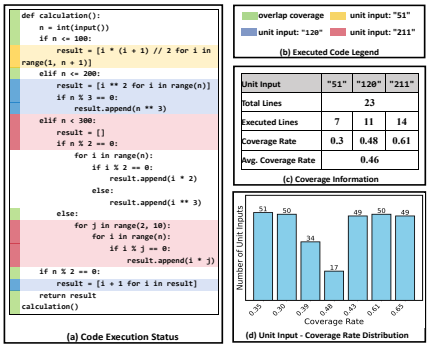

Figure 9: Reports of unit test analysis.

we have intentionally introduced some code smell patterns and vulnerabilities into the code. In Figure 7, the yellow warning boxes indicate the locations where MPLSandbox has detected code smells, while the red warning boxes mark the positions of identified code bugs.

Code Efficient Evaluation provides an analysis of code execution efficiency for different test cases. Figure 8 reports the Hits (the number of times a code line is executed), Time (the total execution time of the code line in milliseconds), Per Hits (the average time required for each execution of the code line in milliseconds), and %Time (the percentage of the total execution time taken by the execution time of the code line). As shown in Figure 8, code lines 2, 3, 5, 22 and 24 have common execution records under different test inputs, with some code lines taking a longer execution time under specific inputs. For example, code line 6 takes 58.1 milliseconds to execute under the input "120" because in this case, line 6 is a loop that iterates 120 times. Code line 23 takes 33.2 milliseconds to execute under the input "210" because this line of code contains a loop that iterates based on the variable result, which is strongly related to the input 210. Code lines 12, 13, and 14 have a large number of executions under the input "210", because this part involves the processing of a large range loop. Therefore, these perceptions of code line execution efficiency undoubtedly provide very important basis for further performance optimization.

Unit Test Analysis returns a comprehensive coverage report for the given unit tests. As shown in Figure 9, green lines represent the overlapping parts of the executed lines for different unit inputs, while yellow, blue, and red lines represent the non-overlapping parts of the executed lines for the test cases "51", "120", and "210", respectively. With the unit input "51", a total of 7 lines of code were executed, achieving a coverage rate of 0.3. For a total of 23 lines of code, the overall average coverage rate is 0.46. This indicates that the current test cases do not fully cover the code paths.

Furthermore, Unit Test Analysis has conducted a complete coverage statistics for all test inputs within the given range. It can be observed that within the range of unit input 0 <= n < 300, this set of code has resulted in 7 different coverage possibilities, with the highest being 0.65 and the lowest being 0.35. The distribution of unit inputs across various coverage rates is relatively even. It is evident that after iterating through all possible test inputs, the code coverage remains at a relatively low level, suggesting that the logical framework of the code itself still has significant room for improvement.