

LiveCANNBench: Benchmark SWE AI Coding for Ascend CANN

Sijie Wang^{1*} Kai Zhao^{1*}

Wee Peng Tay¹ Shuo Zhang² Chengwen Liu³ Quanjiang Guo⁴ Ren Junhao¹ Xin Li¹
Heng Lian⁵ Jingdi Lei¹ Rui She^{6†} Huacan Wang^{7†} Ronghao Chen^{7,8†}

¹NTU ²BUPT ³LZU ⁴UESTC ⁵XDU ⁶BUAA ⁷QuantaAlpha ⁸IAPM (Guangdong)

*Equal Contribution †Corresponding Author

<https://github.com/sijieaaa/LiveCANNBench>

Abstract

AI coding has emerged as a core application of large language models (LLMs), evolving from single-file coding tasks towards complex software engineering (SWE) scenarios. Recent advances in agents have enabled multi-file, multi-language, and dependency-aware AI coding, significantly expanding the scope of AI-assisted software development. While a variety of benchmarks have been proposed to evaluate coding capabilities in general-purpose or GPU coding ecosystems such as CUDA and ROCm, systematic evaluation for Huawei Ascend CANN remains largely underexplored. In this work, we propose **LiveCANNBench**, an SWE-level benchmark designed for AI coding in the CANN software stack. LiveCANNBench is constructed from real-world CANN repositories and consists of over 400 task instances spanning multi-file, multi-language, and execution-aware coding challenges. Unlike existing static benchmarks that primarily focus on kernel-level code generation, LiveCANNBench adopts a live benchmarking paradigm, effectively mitigating data leakage and enabling more reliable evaluation of modern coding agents.

1 Introduction

AI coding has rapidly emerged as one of the core applications of large language models (LLMs) in recent years. Early studies primarily focus on single-file and LeetCode-style coding tasks, which are significantly simplified compared to real-world development and operations (DevOps) scenarios. With the advancement of LLM capabilities, particularly in tool use and multi-turn interaction, agent-based systems have demonstrated the ability to tackle coding problems at the software engineering (SWE) level. At this level, a typical coding task involves multiple files, complex dependency structures, and different coding languages, posing substantially greater challenges than single-file set-

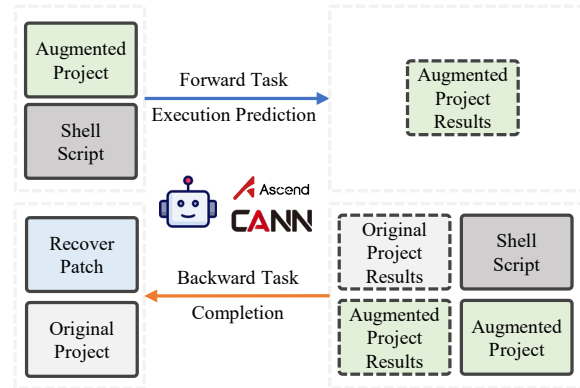


Figure 1: A simplified visualization of the task pipeline. The forward task focuses on predicting execution results from code projects, while the backward task reverses this process by inferring code updates from execution outcomes.

tings. To address these challenges, a wide range of SWE-oriented coding agents and frameworks have been proposed (Yang et al., 2024; Wang et al., 2024; Gao et al., 2025; Xia et al., 2025).

To verify the coding ability of different models, benchmarks are essential. Previously, benchmarks (Zhuo et al., 2024; Jain et al., 2024; Chen et al., 2021; Austin et al., 2021) have been proposed to focus on simple coding tasks. Later, the advancement of coding agents has also promoted an increasing number of SWE benchmarks. More recently, the rapid progress of LLMs has extended beyond general-purpose coding to specialized domains. In particular, the growing demand for AI assistant tools in GPU¹ coding scenarios has highlighted the need for domain-specific evaluation. Multiple GPU software ecosystems coexist in practice, including NVIDIA CUDA, AMD ROCm, and Huawei Ascend CANN, along with higher-level abstractions such as Triton.

¹For illustration simplicity and clarity, we use the term GPU throughout this paper to broadly denote GPU-like hardware accelerators.

For CUDA and ROCm, benefiting from well-established software ecosystems, an increasing number of benchmarks have been proposed to evaluate coding capabilities in these environments (Ouyang et al., 2025; Li et al., 2025a; Kalade and Schelle, 2025; Heakl et al., 2025). These benchmarks cover a variety of GPU coding tasks and have facilitated systematic evaluation of model performance in mature GPU ecosystems.

By contrast, benchmarks targeting the Huawei Ascend CANN ecosystem remain relatively underexplored. Existing works such as NPUKernelBench² and MultiKernelBench (Wen et al., 2025) are designed to support CANN, but they are largely limited to GPU kernel generation tasks, leaving a wide range of practical coding scenarios underexplored. Moreover, these benchmarks adopt static evaluation settings, which may introduce data leakage risks and consequently compromise the fairness and robustness of benchmarking results.

To address these limitations, we propose **LiveCANNBench**, a live and comprehensive benchmark designed to evaluate AI coding capabilities within the CANN software stack. Specifically, LiveCANNBench is built upon official CANN repositories that encompasses multiple coding languages, such as Python, C++, and Shell. LiveCANNBench comprises over 400 task instances constructed from real-world projects, all at the multi-file project level to closely reflect real-world SWE scenarios. We define two complementary task types: (1) forward tasks that require predicting execution results from code, and (2) backward tasks that require generating code from execution results. This dual-task design enables more comprehensive evaluation of both code understanding and generation capabilities. Furthermore, we construct isolated container environments for benchmarking, enabling safer and more reliable evaluation.

2 Related Work

In this section, we provide an overview of related works on AI coding benchmarks.

2.1 General-Purpose Coding Benchmark

At the early stage of AI coding, numerous benchmarks are proposed to evaluate models on relatively simple coding tasks. In particular, many benchmarks focus on single-language settings, such as

HumanEval (Chen et al., 2021), MBPP (Austin et al., 2021), CRUXEval (Gu et al., 2024), and Big-CodeBench (Zhuo et al., 2024), along with their improved variants (Yu et al., 2024; Liu et al., 2023a).

Beyond single-language benchmarks, some efforts extend evaluation to multilingual coding settings. For example, CrossCodeEval (Ding et al., 2023) covers 4 coding languages, while MultiPL-E (Cassano et al., 2022) significantly expands this scope to 19 coding languages, enabling more comprehensive assessments of cross-lingual coding capabilities.

In addition, several benchmarks aim to mitigate data leakage by collecting live or continuously updated coding problems from online sources. Representative examples include LiveCodeBench (Jain et al., 2024) and USACO (Shi et al., 2024), which evaluate models on problems that are less likely to appear in their training corpora.

With the rapid advancement of code models, increasingly strong project-level coding capabilities have emerged, motivating the development of benchmarks that focus on realistic SWE problems (Jimenez et al., 2023; Liu et al., 2023b; Zhang et al., 2025), rather than isolated single-file tasks. A representative example is SWE-Bench, which evaluates models on repository-level bug-fixing tasks that require understanding multiple files, dependencies, and real-world development workflows.

2.2 GPU Coding Benchmark

Having achieved strong performance on general-purpose coding benchmarks, coding agents have been able to tackle more challenging tasks in GPU-related coding. KernelBench (Ouyang et al., 2025) and HPCTransCompile (Lv et al., 2025) focus on benchmarking GPU kernel development within the CUDA ecosystem, evaluating models’ ability to generate efficient and correct CUDA kernels. Beyond CUDA, several works extend evaluation to AMD platforms, such as Geak (Wang et al., 2025a) and NPUEval (Kalade and Schelle, 2025). In addition, CASS (Heakl et al., 2025) and HeteroBench (Tian et al., 2025) further broaden the scope by assessing coding performance across multiple heterogeneous GPU platforms.

In contrast, benchmarking for Huawei Ascend remain relatively underexplored. Existing benchmarks, such as NPUKernelBench and MultiKernelBench (Wen et al., 2025), are primarily restricted to kernel generation tasks for Ascend GPUs. Moreover, these benchmarks are designed for traditional

²<https://openi.pcl.ac.cn/PCL-Benchmark/NPUKernelBench>

Benchmark	#Task Instances	Generated Languages	Project-Level	Agent	Container Environment
NPUKernelBench	157	C/C++ Python	×	×	×
MultiKernelBench	285	C/C++ Python	×	×	×
LiveCANNBench	414	C/C++ Header Python CMake Shell	✓	✓	✓

Table 1: Comparison between existing CANN coding benchmarks and LiveCANNBench.

LLM-based evaluation, without supporting more advanced agent-level settings that involve multi-turn interactions and tool usage.

By comparison as shown in Table 1, LiveCANNBench supports project-level and agent-level benchmarking, providing a closer approximation to real-world SWE scenarios. In such settings, coding agents operate on multi-file projects with diverse dependencies and coding languages, while interacting environments with tools. Thus LiveCANNBench could offer a more realistic and comprehensive evaluation of AI coding capabilities in the Ascend ecosystem.

3 LiveCANNBench

In this section, we provide a more detailed introduction to LiveCANNBench, covering data preparation and task formulation.

3.1 Data Preparing

3.1.1 Repository Collection

For the CANN ecosystem, Huawei provides a collection of official repositories hosted on Gitee³, which serve as a natural starting point for constructing our benchmark. However, the relatively less well-maintained software ecosystem limits the practical usability of many repositories, i.e., a substantial portion of them cannot be successfully deployed or executed in practice. In addition, some repositories do not provide complete executable projects or runnable examples, which further prevents them from being used for the benchmark.

Therefore, we selectively curate 5 well-maintained and easy-to-use repositories that also contain sufficient executable projects and examples.

3.1.2 Shell Script Preparing

The task instances are constructed from the executable projects or usage examples contained within the selected repositories. For example, for

³<https://gitee.com/ascend>

"ascendc-api-adv", it provides a rich set of project-level usage examples, ranging from matrix operations to non-linear activation functions.

An overview of the task instance construction process is illustrated in Fig. 2. To systematically capture the low-level runtime behavior of each project, we first formalize its execution procedure into a standalone shell script that encapsulates all necessary steps, such as environment initialization and command invocation. We then verify the correctness of the project by checking whether the shell script can be executed successfully.

3.1.3 Build Task Instances

Later, we build the task instances, where the pipeline demo is shown in Fig. 2. First, we execute the shell script using the strace tool, a widely adopted diagnostic utility that records all system calls made by the script’s process. With the tool, we identify and collect necessary project files that are accessed during the execution.

We then augment each traced file from the project. Specifically, for each traced file, we randomly select a contiguous code block with a maximum of 3 lines and replace it with a special text mark, "SPECIAL_MARK". This modification also yields the corresponding patch. Afterward, we re-execute the shell script to obtain the augmented execution results, and patches that produce excessively long outputs or do not trigger errors are filtered out.

We apply the above augmentation process to each traced file across all projects. As a result, each code project can generate multiple augmented patches with corresponding execution results, and each augmented patch corresponds to 2 task instances, which will be introduced later.

3.2 Task Formulation

We can categorize coding tasks into forward tasks and backward tasks (Li et al., 2025b; Wang et al., 2025b). In the forward task, the code model is required to predict the execution results of running a shell script. To achieve this goal, the model must

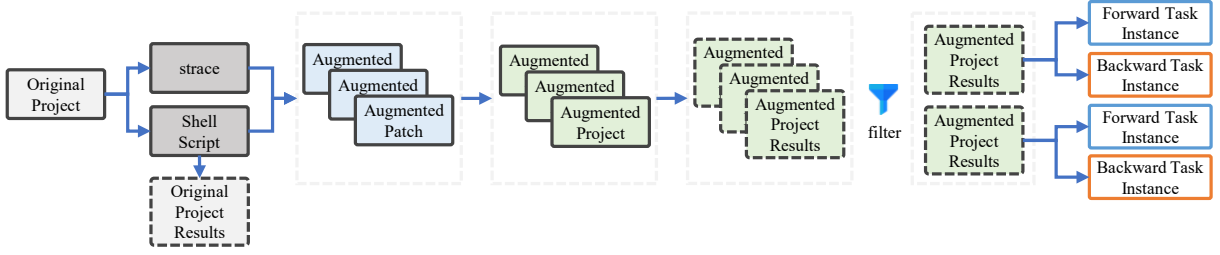


Figure 2: Visualization of the creation of task instances. Each original project can produce multiple augmented patches, and each filtered patch corresponds to a forward and a backward task instances.

fully understand the structure of the entire code project and accurately reason its execution behavior. Conversely, it is natural to consider the backward task, which represents the reverse process. In this setting, the execution results are provided, and the goal is to complete the code project accordingly. To succeed, the model needs to reversely reason the project and generate compatible code completion patches that satisfy the given execution outcomes.

Formally, we denote a task instance as:

$$T = (D, D_{\text{aug}}, P, S, O, O_{\text{aug}}). \quad (1)$$

Here, D and D_{aug} denote the original and augmented projects, respectively. P is the patch that transforms the original project D into the augmented project D_{aug} , while P^{-1} denotes the corresponding inverse patch that restores D_{aug} to D . S is the shell script file. O and O_{aug} are the corresponding execution results obtained from D and D_{aug} , respectively. We formulate the relationships as follows:

$$D_{\text{aug}} = g_{\text{apply}}(D, P), \quad (2)$$

$$D = g_{\text{apply}}(D_{\text{aug}}, P^{-1}), \quad (3)$$

$$O = g_{\text{exec}}(D, S), \quad (4)$$

$$O_{\text{aug}} = g_{\text{exec}}(D_{\text{aug}}, S), \quad (5)$$

where g_{apply} denotes the patch application function, and g_{exec} denotes the shell script execution function.

3.2.1 Forward Task: Execution Prediction

The forward task solving process f_{for} can be formulated as:

$$f_{\text{for}} : (D_{\text{aug}}, S) \mapsto \hat{O}_{\text{aug}}. \quad (6)$$

In this task setting, the code model is provided with the augmented code project D_{aug} and the shell script file S , and the objective is to predict the corresponding execution output \hat{O}_{aug} . A simplified

example of the prompt and the agent’s response is illustrated in Fig. 3.

Evaluation is performed by comparing the predicted execution results \hat{O}_{aug} with the ground-truth execution results O_{aug} . Specifically, we adopt the Levenshtein similarity (edit similarity, ES), following common practice in existing AI coding benchmarks. In addition, we compute both line-level and character-level variants of the metric, denoted as ES-L and ES-C, respectively. These two scores capture output similarity at different levels of granularity.

3.2.2 Backward Task: Completion

The backward task is a completion task, which can be formulated as:

$$f_{\text{back}} : (D_{\text{aug}}, S, O, O_{\text{aug}}) \mapsto \hat{P}^{-1}. \quad (7)$$

In the backward task, the code model is given with the augmented code project D_{aug} , the shell script file S , and the execution result O_{aug} and O . The goal is to predict a recover patch \hat{P}^{-1} , which can recover the code project D_{aug} to \hat{D} , such that \hat{O} and O are as close as possible. An example is shown in Fig. 3.

Since the backward task requires predicting patches, we introduce patch format validity as an additional evaluation criterion. Moreover, similar to the evaluation protocol described in Section 3.2.1, we compute both ES-L and ES-C based on the predicted execution results \hat{O}_{org} and the ground-truth results O . Different from the forward task, predicted patches with invalid formats are assigned zero scores for both ES-L and ES-C.

3.3 Statistics

In the current benchmark version, we provide a total of 414 task instances, collected from official CANN repositories committed between 2025-09 and 2025-12. Statistics on the number of task instances are visualized in Fig. 4. Each repository

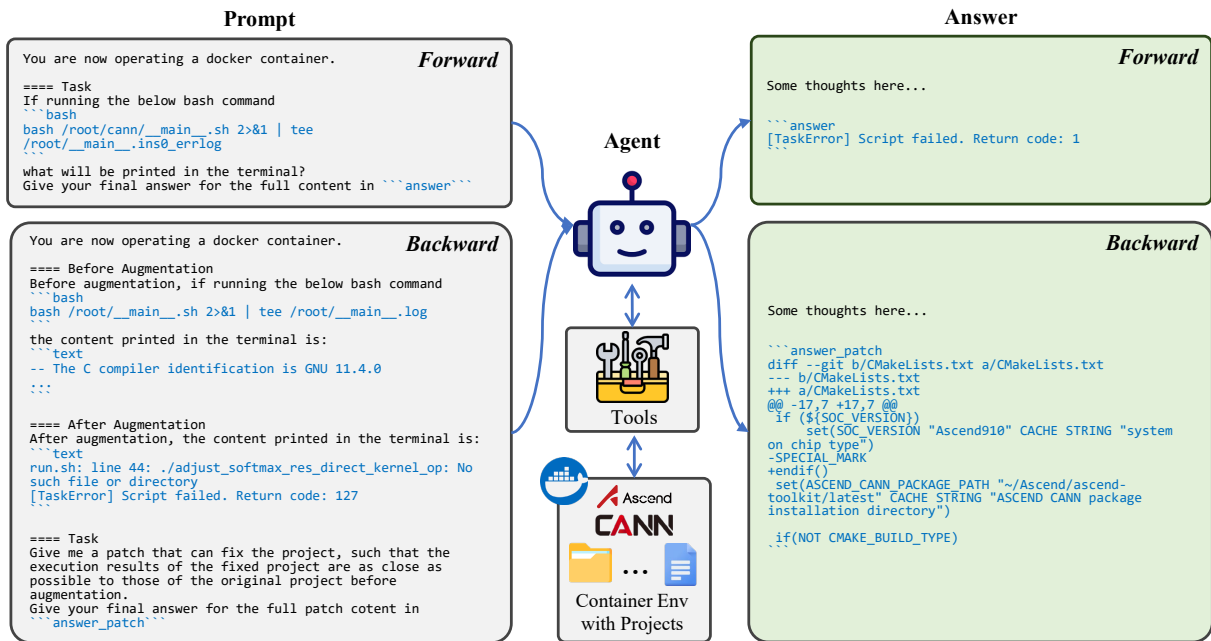


Figure 3: Overview structure and simplified task prompts for both forward and backward tasks. The task environments are packed into Docker containers, and agents can use tools to interact with the task environments.

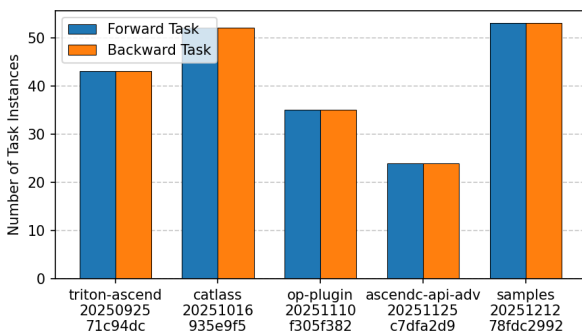


Figure 4: Number of task instances in different repositories. All repositories have at least 40 task instances.

contributes at least 40 task instances, ensuring sufficient and balanced evaluation coverage across repositories.

On the other hand, we also present statistics on the distribution of coding language types, as shown in Fig. 5. Overall, Python, C, C++, and CMake constitute the largest proportion among all language types. The "samples" repository exhibits the most diverse language composition, containing 10 prominent coding languages.

We further report statistics on the execution result lengths of task instances from different repositories in Table 2. The "samples" repository produces the longest execution results, with character lengths exceeding 9600 and line lengths over 100. Regarding the augmented code blocks, all repositories have an average augmentation length of 1.8

lines.

4 Experiment

In this section, we present implementation details, experimental results, and performance analysis.

4.1 Implementation Details

We evaluate the performance of two representative agent frameworks, including LangChain⁴ and mini-SWE-Agent (Yang et al., 2024). LangChain is a general-purpose agent framework that supports customized tool configurations. In our setting, We configure it with read-only file-explorer tools, which can be regarded as the non-intrusive coding agent.

mini-SWE-Agent specializes for AI coding and is equipped with a terminal command execution tool with full permissions. We do not modify the settings and keep them as default. Note that we do not test mini-SWE-Agent in forward tasks, as it can easily obtain execution results with the terminal tool, which is trivial and meaningless in evaluation.

We test several LLMs (including both open-source and proprietary models) as the chat models in agents. The used LLMs consist of GPT-5-mini (OpenAI, 2025), DeepSeek-V3.1-terminus (DeepSeek, 2025), Gemini-2.5-flash (Comanici et al., 2025), Grok-4.1-fast-non-reasoning (xAI, 2025), GLM-4.6 (GLM, 2025),

⁴<https://github.com/langchain-ai/langchain>

Statistics / Repository	triton-ascned 20250925 71c94dc	catlass 20251016 935e9f5	op-plugin 20251110 f305f382	ascendc-api-adv 20251125 c7dfa2d9	samples 20251212 78fdc2992	Average
Avg. character length of O	411	3310	327	5475	9609	3826
Avg. line length of O	6	45	4.8	74	126	51
Avg. augmented lines in P	2.0	1.7	1.7	1.7	1.8	1.8

Table 2: Statistics on character length of O , line length O , and the number of augmented lines in P .

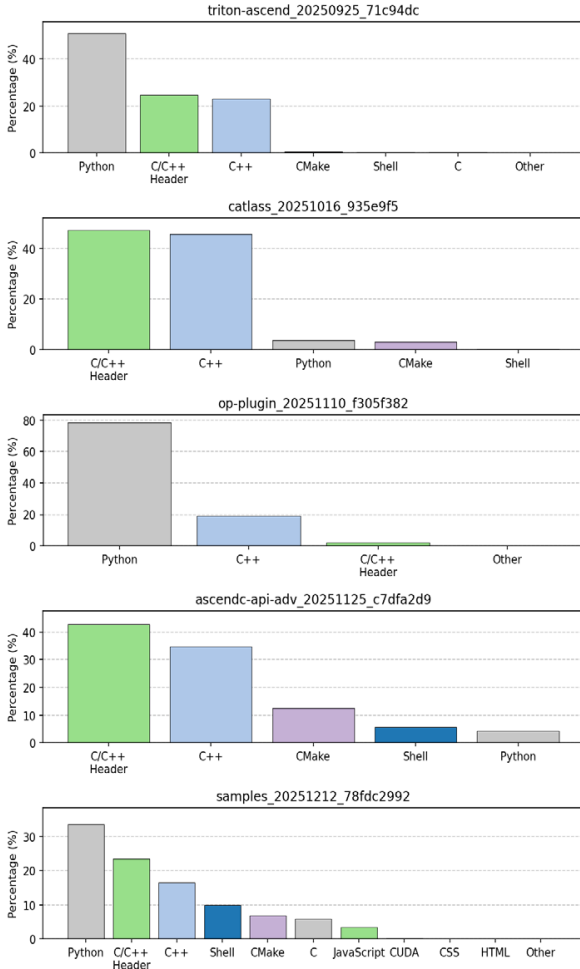


Figure 5: Statistics of coding language distributions in LiveCANNBench. Python and C++ are the dominant languages.

MiniMax-M2 (MiniMax, 2025), and Qwen3-Coder-480B-A35B-Instruct (Yang et al., 2025).

All experiments are conducted on a Ascend 910B GPU, which is the most often used and software-friendly among all Huawei Ascend GPUs. The container environments are built using Docker.

4.2 Performance Comparison

We evaluate the performance of both LangChain and mini-SWE-Agent across different LLMs. As shown in Table 3, both forward and backward tasks remain challenging, with ES-C and ES-L scores

consistently below 50. We further observe that ES-L, which requires exact line-level matching, is consistently more challenging than ES-C, which only considers character-level similarity. This gap indicates that, in SWE-level coding tasks, models may generate semantically similar or partially correct answers at the character level, but they still fail to precisely infer the correct line-level structure.

In the forward task, GPT-5-mini achieves the highest ES-C score, whereas MiniMax-M2 surpasses other models on ES-L. By contrast, Grok-4.1-fast-non-reasoning performs the worst under this setting, with both ES-C and ES-L below 10.

In the backward task, mini-SWE-Agent equipped with full-permission terminal tools consistently outperforms LangChain with read-only tools, demonstrating the effectiveness of SWE-specific agent configurations over general-purpose setups. Among the evaluated LLMs, DeepSeek-V3.1-terminus achieves the highest scores under both agent settings. However, it does not consistently produce format-valid patches, with patch format validity remaining at approximately 60, highlighting the inherent challenges of SWE-level coding for CANN projects.

Interestingly, although GPT-5-mini achieves high scores in the forward task, it fails to produce format-valid patch outputs in the backward task. This observation reveals limitations in its SWE-oriented coding capability and highlights its restricted generalization under customized agent configurations. Moreover, a similar trend can be observed in the official SWE-Bench-Verified benchmark, where the top-performing models, MiniMax-M2 and GPT-5-mini, both show significant performance decreasing in forward and backward tasks. These results suggest that strong performance on specific benchmarks does not necessarily translate to robust generalization across different task formulations or agent settings.

LLM	Forward		Backward			SWE-Bench-Verified Resolved \uparrow
	ES-C \uparrow	ES-L \uparrow	ES-C \uparrow	ES-L \uparrow	Patch Format \uparrow	
LangChain w/ read-only file-explore tools						
DeepSeek-V3.1-terminus	14.9	11.5	16.5	14.6	21.7	-
Gemini-2.5-flash	13.9	8.3	5.7	4.7	7.6	-
GPT-5-mini	21.2	12.2	0	0	0	-
Grok-4.1-fast-non-reasoning	7.3	4.9	0.6	0.4	1.2	-
MiniMax-M2	20.8	14.3	1.0	0.8	2.0	-
Qwen3-Coder-480B-A35B-Instruct	15.8	13.0	2.6	2.3	4.7	-
GLM-4.6	19.7	9.8	10.9	9.7	13.5	-
mini-SWE-Agent w/ full-permission terminal tools (default)						
DeepSeek-V3.1-terminus	-	-	49.4	43.1	59.2	-
Gemini-2.5-flash	-	-	23.1	20.1	34.7	-
GPT-5-mini	-	-	20.7	16.8	39.7	59.8*
Grok-4.1-fast-non-reasoning	-	-	9.6	8.5	14.5	-
MiniMax-M2	-	-	17.0	15.1	19.0	61.0*
Qwen3-Coder-480B-A35B-Instruct	-	-	39.5	34.6	46.2	55.4*
GLM-4.6	-	-	47.1	41.6	62.2	55.4*

Table 3: Performance comparison of different agent-LLM combinations. The best score in each metric is highlighted with bold fonts. LangChain is with customized read-only tools, which can represent non-intrusive coding scenarios, while mini-SWE-Agent with default terminal tools can represent intrusive coding scenarios. "-" denotes the result is unavailable. "Patch Format" denotes the patch format validity. The scores with "*" are from the official leaderboard.

LLM	#LLM Calls \downarrow	Forward		#LLM Calls \downarrow	Backward	
		#In Tokens \downarrow	#Out Tokens \downarrow		#In Tokens \downarrow	#Out Tokens \downarrow
LangChain w/ read-only file-explore tools						
DeepSeek-V3.1-terminus	17.6	203.9k	3.4k	20.5	305.5k	5.3k
Gemini-2.5-flash	5.1	25.6k	5.4k	2.9	22.9k	2.5k
GPT-5-mini	10.9	79.8k	5.4k	6.2	57.3k	2.5k
Grok-4.1-fast-non-reasoning	10.3	171.8k	2.6k	5.9	51.6k	1.4k
MiniMax-M2	20.3	262.8k	7.8k	19.2	276.0k	7.3k
Qwen3-Coder-480B-A35B-Instruct	12.9	112.9k	3.0k	9.0	91.8k	2.1k
GLM-4.6	17.3	234.1k	2.9k	17.7	238.3k	3.7k
mini-SWE-Agent w/ full-permission terminal tools (default)						
DeepSeek-V3.1-terminus	-	-	-	19.2	169.1k	3.2k
Gemini-2.5-flash	-	-	-	9.7	112.8k	7.1k
GPT-5-mini	-	-	-	6.2	47.4k	8.8k
Grok-4.1-fast-non-reasoning	-	-	-	20.5	159.0k	3.3k
MiniMax-M2	-	-	-	25.6	282.9k	6.8k
Qwen3-Coder-480B-A35B-Instruct	-	-	-	22.7	225.7k	3.3k
GLM-4.6	-	-	-	24.9	254.2k	3.4k

Table 4: Usage comparison of different agent-LLM combinations. "#LLM Calls" indicates the number of LLM API invocations used to solve the task instance. Input and output tokens are accumulated over the entire task-solving process for each instance.

4.3 Usage Comparison

Besides scoring higher in the benchmark, the LLM calling and token usage in different agents are also important considerations. In SWE coding tasks, the agent needs to interact with the environments to acquire necessary information and infer the results, which is different from that in single-file coding problems. As shown in Table 4, agents generally consume more LLM calls and tokens in the forward tasks, reflecting the increased interaction and

reasoning overhead required in this setting.

4.4 Repository Comparison

We further report the detailed performance of each repository as illustrated in Table 5. Among all evaluated repositories, "samples" emerges as the easiest benchmark. It gives the highest scores in ES-C, ES-L, and patch format validity, while also requiring the fewest output tokens. Interestingly, as visualized in Fig. 5, the "samples" repository contains

Repository	ES-C \uparrow	ES-L \uparrow	Patch Format \uparrow	#LLM Calls \downarrow	#In Tokens \downarrow	#Out Tokens \downarrow	#Stars	Initial Release
triton-ascend_20250925_71c94dc	39.4	36.6	41.9	19.7	107.8k	3.4k	0.1k	2025-05
catlass_20251016_935e9f5	38.5	37.6	46.2	21.4	154.1k	3.6k	0.2k	2025-01
op-plugin_20251110_f305f382	43.0	35.8	54.3	20.6	125.3k	3.2k	0.1k	2023-04
ascendc-api-adv_2251125_c7dfa2d9	53.0	45.4	70.7	19.3	241.5k	3.4k	0.1k	2024-07
samples_20251212_78fdc2992	72.9	60.3	83.0	14.7	216.6k	2.5k	1.3k	2018-01

Table 5: The performance in different repositories in backward tasks. mini-SWE-Agent with DeepSeek-V3.1-terminus is used here. The number of stars is counted by the end of the submission date.

the most diverse set of coding languages. This observation suggests that language diversity is not necessarily correlated with task difficulty.

In contrast, "triton-ascend", which primarily adopts the Triton coding language, is a challenging repository. This increased difficulty can be largely attributed to the relative novelty of the Triton language and its limited maturity within the CANN ecosystem. This interpretation is further supported by repository metadata. As shown in Table 5, "triton-ascend" has a small number of stars and the latest release date. By contrast, "samples" has a large number of stars and the earliest release date, implying a higher likelihood of being included in the LLM training corpora.

Finally, we observe a clear relationship between repository difficulty and agent interaction cost. The easiest repository, "samples", requires the fewest LLM calls, whereas more challenging repositories such as "triton-ascend", "catlass", and "op-plugin" consume substantially more LLM calls. This indicates that, for harder tasks, the agent needs to perform more iterative interactions with the code project before producing the answer.

4.5 Augmentation Comparison

As described in Section 3, we construct augmented projects using line-level augmentation. We further analyze the impact of different augmentation levels on task difficulty. As in Fig. 6, higher augmentation levels lead to substantially more challenging tasks. With 1-line augmentation, ES-C, ES-L, and patch format validity all achieve scores near 50. In contrast, increasing the augmentation to 3 lines results in scores falling below 30.

5 Limitation

In LiveCANNBench, all task instances constructed from official CANN repositories. While this design ensures correctness and stability, an increasing number of community-driven projects have also

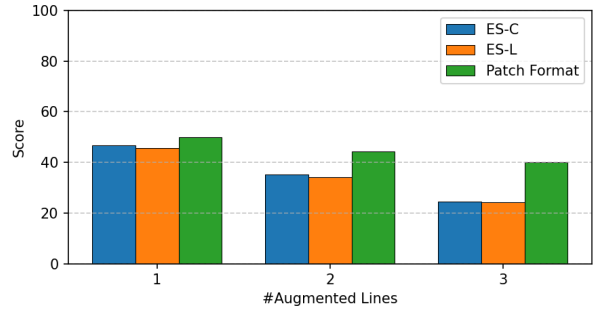


Figure 6: Performance comparison with different augmentation levels on "catlass". mini-SWE-Agent with DeepSeek-V3.1-terminus is used here.

adopted CANN. Incorporating a broader range of CANN-based projects could further improve the diversity of LiveCANNBench. Moreover, the current benchmark focuses on Ascend 910B devices, which currently provide the most complete and stable open-source support in the CANN ecosystem. Extending LiveCANNBench to additional Ascend devices would allow for a more comprehensive evaluation of hardware-aware and cross-device generalization.

6 Conclusion

In this work, we propose LiveCANNBench, an SWE-level benchmark designed to evaluate AI coding capabilities within the CANN ecosystem. LiveCANNBench comprises over 400 task instances curated from real-world CANN projects, covering a broad spectrum of hardware-related, multi-file, and multi-language coding challenges. These tasks provide a realistic and challenging testbed for assessing the effectiveness and robustness of AI coding agents.

7 Acknowledgement

This work was partially supported by the Fundamental Research Funds for the Central Universities and the National Natural Science Foundation of China under Grant No. 62576025.

References

- Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and 1 others. 2021. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*.
- Federico Cassano, John Gouwar, Daniel Nguyen, Sydney Nguyen, Luna Phipps-Costin, Donald Pinckney, Ming-Ho Yee, Yangtian Zi, Carolyn Jane Anderson, Molly Q Feldman, and 1 others. 2022. Multiple: 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, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, and 39 others. 2021. [Evaluating large language models trained on code](#). *Preprint*, arXiv:2107.03374.
- Gheorghe Comanici, Eric Bieber, Mike Schaekermann, Ice Pasupat, Noveen Sachdeva, Inderjit Dhillon, Marcel Blistein, Ori Ram, Dan Zhang, Evan Rosen, and 1 others. 2025. Gemini 2.5: Pushing the frontier with advanced reasoning, multimodality, long context, and next generation agentic capabilities. *arXiv preprint arXiv:2507.06261*.
- DeepSeek. 2025. Deepseek-v3.1-terminus. <https://api-docs.deepseek.com/news/news250922/>.
- Yangruibo Ding, Zijian Wang, Wasi Ahmad, Hantian Ding, Ming Tan, Nihal Jain, Murali Krishna Ramathan, Ramesh Nallapati, Parminder Bhatia, Dan Roth, and 1 others. 2023. Crosscodeeval: A diverse and multilingual benchmark for cross-file code completion. *Advances in Neural Information Processing Systems*, 36:46701–46723.
- Pengfei Gao, Zhao Tian, Xiangxin Meng, Xinchun Wang, Ruida Hu, Yuanan Xiao, Yizhou Liu, Zhao Zhang, Junjie Chen, Cuiyun Gao, and 1 others. 2025. Trae agent: An llm-based agent for software engineering with test-time scaling. *arXiv preprint arXiv:2507.23370*.
- GLM. 2025. Glm-4.6. <https://z.ai/blog/glm-4.6/>.
- Alex Gu, Baptiste Rozière, Hugh Leather, Armando Solar-Lezama, Gabriel Synnaeve, and Sida I Wang. 2024. Cruxeval: A benchmark for code reasoning, understanding and execution. *arXiv preprint arXiv:2401.03065*.
- Ahmed Heakl, Sarim Hashmi, Gustavo Bertolo Stahl, Seung Hun Eddie Han, Salman Khan, and Abdulrahman Mahmoud. 2025. Cass: Nvidia to amd transpilation with data, models, and benchmark. *arXiv preprint arXiv:2505.16968*.
- Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. 2024. Livecodebench: Holistic and contamination free evaluation of large language models for code. *arXiv preprint arXiv:2403.07974*.
- Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2023. Swe-bench: Can language models resolve real-world github issues? *arXiv preprint arXiv:2310.06770*.
- Sarunas Kalade and Graham Schelle. 2025. Npueval: Optimizing npu kernels with llms and open source compilers. *arXiv preprint arXiv:2507.14403*.
- Jianling Li, Shangzhan Li, Zhenye Gao, Qi Shi, Yuxuan Li, Zefan Wang, Jiacheng Huang, WangHaojie WangHaojie, Jianrong Wang, Xu Han, and 1 others. 2025a. Tritonbench: Benchmarking large language model capabilities for generating triton operators. In *Findings of the Association for Computational Linguistics: ACL 2025*, pages 23053–23066.
- Junlong Li, Daya Guo, Dejian Yang, Runxin Xu, Yu Wu, and Junxian He. 2025b. Code/o: Condensing reasoning patterns via code input-output prediction. *arXiv preprint arXiv:2502.07316*.
- Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2023a. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. *Advances in Neural Information Processing Systems*, 36:21558–21572.
- Tianyang Liu, Canwen Xu, and Julian McAuley. 2023b. Repobench: Benchmarking repository-level code auto-completion systems. *arXiv preprint arXiv:2306.03091*.
- Jiaqi Lv, Xufeng He, Yanchen Liu, Xu Dai, Aocheng Shen, Yinghao Li, Jiachen Hao, Jianrong Ding, Yang Hu, and Shouyi Yin. 2025. Hpctranscompile: An ai compiler generated dataset for high-performance cuda transpilation and llm preliminary exploration. *arXiv preprint arXiv:2506.10401*.
- MiniMax. 2025. Minimax-m2. <https://www.minimax.io/news/minimax-m2/>.
- OpenAI. 2025. Introducing gpt-5. <https://openai.com/index/introducing-gpt-5/>.
- Anne Ouyang, Simon Guo, Simran Arora, Alex L Zhang, William Hu, Christopher Ré, and Azalia Mirhoseini. 2025. Kernelbench: Can llms write efficient gpu kernels? *arXiv preprint arXiv:2502.10517*.
- Quan Shi, Michael Tang, Karthik Narasimhan, and Shunyu Yao. 2024. Can language models solve olympiad programming? *arXiv preprint arXiv:2404.10952*.

- Hongzheng Tian, Alok Mishra, Zhiheng Chen, Rolando P Hong Enriquez, Dejan Milojicic, Eitan Frachtenberg, and Sitao Huang. 2025. Heterobench: Multi-kernel benchmarks for heterogeneous systems. In *Proceedings of the 16th ACM/SPEC International Conference on Performance Engineering*, pages 320–333.
- Jianghui Wang, Vinay Joshi, Saptarshi Majumder, Xu Chao, Bin Ding, Ziqiong Liu, Pratik Prabhakaran Brahma, Dong Li, Zicheng Liu, and Emad Barsoum. 2025a. Geak: Introducing triton kernel ai agent & evaluation benchmarks. *arXiv preprint arXiv:2507.23194*.
- Sijie Wang, Quanjiang Guo, Kai Zhao, Yawei Zhang, Xin Li, Xiang Li, Siqi Li, Rui She, Shangshu Yu, and Wee Peng Tay. 2025b. Codeboost: Boosting code llms by squeezing knowledge from code snippets with rl. *arXiv preprint arXiv:2508.05242*.
- Xingyao Wang, Boxuan Li, Yufan Song, Frank F Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, and 1 others. 2024. Openhands: An open platform for ai software developers as generalist agents. *arXiv preprint arXiv:2407.16741*.
- Zhongzhen Wen, Yinghui Zhang, Zhong Li, Zhongxin Liu, Linna Xie, and Tian Zhang. 2025. Multikernelbench: A multi-platform benchmark for kernel generation. *arXiv eprints*, pp. *arXiv-2507*.
- xAI. 2025. Grok 4.1. <https://x.ai/news/grok-4-1/>.
- Chunqiu Steven Xia, Zhe Wang, Yan Yang, Yuxiang Wei, and Lingming Zhang. 2025. Live-swe-agent: Can software engineering agents self-evolve on the fly? *arXiv preprint arXiv:2511.13646*.
- An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, and 1 others. 2025. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*.
- John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. 2024. Swe-agent: Agent-computer interfaces enable automated software engineering. *Advances in Neural Information Processing Systems*, 37:50528–50652.
- Zhaojian Yu, Yilun Zhao, Arman Cohan, and Xiaoping Zhang. 2024. Humaneval pro and mbpp pro: Evaluating large language models on self-invoking code generation. *arXiv preprint arXiv:2412.21199*.
- Linghao Zhang, Shilin He, Chaoyun Zhang, Yu Kang, Bowen Li, Chengxing Xie, Junhao Wang, Maoquan Wang, Yufan Huang, Shengyu Fu, and 1 others. 2025. Swe-bench goes live! *arXiv preprint arXiv:2505.23419*.
- Terry Yue Zhuo, Minh Chien Vu, Jenny Chim, Han Hu, Wenhao Yu, Ratnadira Widayarsi, Imam Nur Bani Yusuf, Haolan Zhan, Junda He, Indraneil Paul, and 1 others. 2024. Bigcodebench: Benchmarking code generation with diverse function calls and complex instructions. *arXiv preprint arXiv:2406.15877*.

S1 More Implementation Details

In this work, all experiments are conducted on Ascend 910B GPUs. The used operation system is Ubuntu 22.04, which is one of the most often used version.

During data preparing, we use the strace tool to track the invoked file. Specifically, the traced file types include Python, C++, C, and CMake.

In the LangChain agent, we use a total of 6 read-only tools, including "get_project_tree", "read_file", "grep_str_in_file", "sed_file", "obtain_absolute_git_dir", and "check_patch". Other configurations are maintained as default.

For both LangChain and mini-SWE-Agent, we set the maximum number of LLM invocations as 30.

S2 Evaluation Metrics

In this work, we adopt two primary evaluation metrics, ES-C and ES-L, both of which are based on edit similarity. Specifically, we compute the normalized Levenshtein similarity using the "levenshtein.normalized_similarity" function from the "textdistance" Python package, which we denote as g_{lev} . The metrics are computed as:

$$ES-C = g_{lev}(\log_1, \log_2), \quad (S8)$$

$$ES-L = g_{lev}(g_{lines}(\log_1), g_{lines}(\log_2)), \quad (S9)$$

where \log_1 and \log_2 denote the execution logs (e.g., $O, O_{aug}, \hat{O}, \hat{O}_{aug}$), and g_{lines} splits a log into a sequence of lines.

S3 Repositories

We give the description of the curated repositories as follows:

- **triton-ascend** (<https://gitee.com/ascend/triton-ascend>). It is a Triton compilation framework designed for the Ascend platform, aiming to enable efficient execution of Triton programs on Ascend hardware.
- **catlass** (<https://gitee.com/ascend/catlass>). This is a project focused on providing high-performance foundational templates for matrix multiplication-related operators.
- **op-plugin** (<https://gitee.com/ascend/op-plugin>). This project develops the

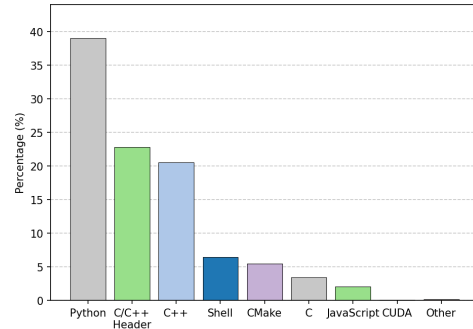


Figure S7: Visualization on coding language statistics in the whole benchmark.

Ascend Extension for PyTorch ("torch_npu") operator plugin, which provides PyTorch developers with convenient access to NPU operator libraries. The compilation and usage of the OpPlugin operator plugin depend on the Ascend Extension for PyTorch.

- **ascendc-api-adv** (<https://gitee.com/ascend/ascendc-api-adv>). This repository contains the source code of the Ascend C high-level APIs. These high-level APIs typically provide abstractions and encapsulations of common algorithms based on single-core implementations. They are usually built upon and implemented by invoking multiple low-level APIs. The high-level APIs include mathematical libraries, MatMul, Softmax, and related APIs.
- **samples** (<https://gitee.com/ascend/samples>). It provides a rich collection of code examples covering a wide range of scenarios, including media data processing (DVPP/AIPP), operator development and invocation (Ascend C), as well as inference application development and deployment (AscendCL).

S4 Overall Language Statistics

The overall language statistics are shown in Fig. S7. Python, C++, Shell, CMake, and C are the dominant coding languages in LiveCANNBench, where Python is the most-used language.

S5 Trajectories

In this section, we provide agent inference trajectories in both forward and backward tasks. The forward task trajectory is shown in Fig. S8. The backward task trajectories are shown in Figs. S9 to S12.

```

User:
You are now operating a docker container. container_name is `PREDDOUT--apiday--deepseek-ai--deepseek-v3.1-terminus--catlass_20251016_935e9f5--examples--
10_grouped_matmul_slice_m_per_token_dequant_COMP--_main_.ins0_patch`.
Some tools are wrapped to help you inspect the container.

==== System
cmake: cmake version 3.22.1
gcc: gcc (Ubuntu 11.4.0-1ubuntu1~22.04.2) 11.4.0
g++: g++ (Ubuntu 11.4.0-1ubuntu1~22.04.2) 11.4.0
npu-smi info:
+-----+
| npu-smi 25.2.0      Version: 25.2.0      |
+-----+
project dir: /root/cann/LiveCANNBench/tasks/tasks_v2_1to3line/catlass_20251016_935e9f5/catlass_20251016_935e9f5--examples--
10_grouped_matmul_slice_m_per_token_dequant_COMP
SOC_VERSION: Ascend910B4
cann-toolkit: 8.2.rc1

==== Task
After augmentation, if running the below bash command
`bash
bash /root/cann/LiveCANNBench/tasks/tasks_v2_1to3line/catlass_20251016_935e9f5/catlass_20251016_935e9f5--examples--
10_grouped_matmul_slice_m_per_token_dequant_COMP/_main_.sh >&1 | tee
/root/cann/LiveCANNBench/tasks/tasks_v2_1to3line/catlass_20251016_935e9f5/catlass_20251016_935e9f5--examples--
10_grouped_matmul_slice_m_per_token_dequant_COMP/_main_.ins0_errlog
`
what will be printed in the terminal?

You must think step by step to find the answer.
Running command is not supported.
The task repo has been perturbed. Thus, it may contains some bugs.

You must think step by step. Give your final answer for the full content as
`answer
your_predicted_content_here`

Below is an example answer from other projects:
`answer
[INFO]: Current compile soc version is Ascend910B4
-- The C compiler identification is GNU 11.4.0-- Detecting C compiler ABI info - done
-- Check for working C compiler: /usr/bin/cc - skipped
[ 2%] Creating directories for 'kernels_precompile'
[ 4%] No download step for 'kernels_precompile'
[ 6%] No update step for 'kernels_precompile'
[ 9%] No patch step for 'kernels_precompile'
(The rest are omitted for brevity. But you cannot omit them in your final answer.)
`

```

```

AI:
I'll help you determine what will be printed when running the given bash command. Let me start by exploring the project structure and understanding what the script does.

First, let me examine the project directory structure:

```

```

Tool Invoke:
(tree)

Tool Return:
/root/cann/LiveCANNBench/tasks/tasks_v2_1to3line/catlass_20251016_935e9f5/catlass_20251016_935e9f5--examples--
10_grouped_matmul_slice_m_per_token_dequant_COMP/\r\n -- _main_.sh\r\n\r\n0 directories, 1 file\r\n"

```

• • •

```

AI:
Let me check if conda is available in the system

```

```

Tool Invoke:
(grep)

Tool Return:
"11-# conda \r\n12:which_conda=$(which conda)\r\n13-source \"${which_conda%/*}/etc/profile.d/conda.sh\"\r\n"

```

```

AI:
Based on my analysis of the script and the system state, I can predict what will be printed when running this command:

1. The script starts with `set -e` which means it will exit immediately on any error
2. It tries to source the Ascend toolkit environment
3. It attempts to find conda and activate the `catlass_20251016_935e9f5` environment
4. It tries to change to a directory that doesn't exist (`/root/cann/ascend_gitee/catlass_20251016_935e9f5`)
5. It tries to run a build script and execute a binary

Given that the repository directory doesn't exist and the conda environment likely doesn't exist either, the script will fail early in the execution.

`answer
#!/bin/bash
set -e
trap 'echo "[TaskError] Script failed. Return code: $?"; exit $?' ERR

SOC_VERSION=Ascend910B4
source /usr/local/Ascend/ascend-toolkit/set_env.sh
repo_name=catlass_20251016_935e9f5

# conda
which_conda=$(which conda)
source "${which_conda%/*}/etc/profile.d/conda.sh"
conda activate $(repo name)
[TaskError] Script failed. Return code: 1
`

The script will fail when trying to activate the conda environment `catlass_20251016_935e9f5` because this environment doesn't exist in the system. The `conda activate` command will return a non-zero exit code, triggering the ERR trap which will print the error message and exit the script.

```

Figure S8: A forward task trajectory example of LangChain (DeepSeek-V3.1-terminus).

```

User:
You are now operating a docker container. container_name is `COMP--apiday--deepseek-ai--deepseek-v3.1-terminus--catlass_20251016_935e9f5--examples--
00_basic_matmul_COMP--_main_.ins0_patch`.
Some tools are wrapped to help you inspect the container.

==== System
----
cmake: cmake version 3.22.1
gcc: gcc (Ubuntu 11.4.0-1ubuntu1~22.04.2) 11.4.0
g++: g++ (Ubuntu 11.4.0-1ubuntu1~22.04.2) 11.4.0
npu-smi info:
+-----+
| npu-smi 25.2.0                Version: 25.2.0                |
+-----+
project dir: /root/cann/LiveCANNBench/tasks/tasks_v2_1to3line/catlass_20251016_935e9f5/catlass_20251016_935e9f5--examples--00_basic_matmul_COMP
SOC_VERSION: Ascend910B4
cann-toolkit: 8.2.rc1

==== Before Augmentation
Before augmentation, if running the below bash command
`bash
bash /root/cann/LiveCANNBench/tasks/tasks_v2_1to3line/catlass_20251016_935e9f5/catlass_20251016_935e9f5--examples--00_basic_matmul_COMP/_main_.sh 2>&1 | tee
/root/cann/LiveCANNBench/tasks/tasks_v2_1to3line/catlass_20251016_935e9f5/catlass_20251016_935e9f5--examples--00_basic_matmul_COMP/_main_.log
`
the content printed in the terminal is:
`text

(CATLASS)
[INFO]building target: 00_basic_matmul_COMP...
NOT build python extension, skip python related dependencies check.
-- Configuring done (0.1s)
-- Generating done (0.2s)
-- Build files have been written to: /root/cann/ascend_gitee_v2/catlass_20251016_935e9f5/build
[ 50%] Building ASCEND object examples/00_basic_matmul_COMP/CMakeFiles/00_basic_matmul_COMP.dir/basic_matmul.cpp.o
[100%] Linking ASCEND executable 00_basic_matmul_COMP
[100%] Built target 00_basic_matmul_COMP
-- Install configuration: "Release"
-- Installing: /root/cann/ascend_gitee_v2/catlass_20251016_935e9f5/output/bin/00_basic_matmul_COMP
[INFO]Target `00_basic_matmul_COMP` built successfully

Compare success.
....

==== During Augmentation
During augmenting, some lines in the following files have been replace with a single `SPECIAL_MARK`:
CMakelists.txt
....
In each file, a contiguous block of up to 2 lines is replaced with a single `SPECIAL_MARK`.

==== After Augmentation
After augmentation, if running the below bash command
`bash
bash /root/cann/LiveCANNBench/tasks/tasks_v2_1to3line/catlass_20251016_935e9f5/catlass_20251016_935e9f5--examples--00_basic_matmul_COMP/_main_.sh 2>&1 | tee
/root/cann/LiveCANNBench/tasks/tasks_v2_1to3line/catlass_20251016_935e9f5/catlass_20251016_935e9f5--examples--00_basic_matmul_COMP/_main_.ins0_errlog
`
the content printed in the terminal is:
`text

(CATLASS)
[INFO]building target: 00_basic_matmul_COMP...
CMake error at examples/00_basic_matmul_COMP/CMakeLists.txt:10:
Parse error. Expected "(", got newline with text "
"
"
-- Configuring incomplete, errors occurred!
[TaskError] Script failed. Return code: 1
....

==== Task
Give me a patch that can fix the project (i.e., replace `SPECIAL_MARK` with the correct content).
Such that the execution results of the fixed project are as close as possible to those of the original project before augmentation.
I will save your patch into the patch file `/root/cann/ascend_gitee_v2/catlass_20251016_935e9f5/examples/00_basic_matmul_COMP/_answer_.patch` and apply it to the
augmented repo to get the correct repo, i.e.,
`cd /root/cann/ascend_gitee_v2/catlass_20251016_935e9f5/examples/00_basic_matmul_COMP && git apply
/root/cann/ascend_gitee_v2/catlass_20251016_935e9f5/examples/00_basic_matmul_COMP/_answer_.patch
The absolute-git-dir of `/root/cann/ascend_gitee_v2/catlass_20251016_935e9f5/examples/00_basic_matmul_COMP` is
`/root/cann/ascend_gitee_v2/catlass_20251016_935e9f5/examples/00_basic_matmul_COMP/.git`.

You must think step by step to find the answer. Give your final answer for the full patch cotent in ``answer_patch``.
E.g.,
`answer_patch
diff --git b/main.cpp a/main.cpp
--- b/main.cpp
+++ a/main.cpp
@@ -20,7 +20,7 @@ int32_t main(int argc, char const *argv[])

// 8 indicates the kernel function will be executed on eight cores.
constexpr uint32_t blockDim = 8;
-SPECIAL_MARK
+ hello_world_do(blockDim, stream);
+ aclrtSynchronizeStream(stream);
....
aclrtDestroyStream(stream);
`

```

Figure S9: A backward task trajectory example of LangChain (DeepSeek-V3.1-terminus) (Prompt).

AI:

I need to analyze the problem step by step. The error message indicates there's a parsing error in `examples/00_basic_matmul_COMP/CMakeLists.txt` at line 10, where it expected "(" but got a newline. This suggests that `SPECIAL_MARK` replaced some content that should have been proper CMake syntax.

Tool Invoke:

(cat)

Tool Return:

```
"# Copyright (c) 2025 Huawei Technologies Co., Ltd.\r\n# This file is a part of the CANN Open Software.\r\n# Licensed under CANN Open Software License Agreement Version 1.0 (the \"License\").\r\n# Please refer to the license for details. You may not use this file except in compliance with the License.\r\n# THIS SOFTWARE IS PROVIDED ON AN \"AS IS\" BASIS, WITHOUT WARRANTIES OF ANY KIND, EITHER EXPRESS OR IMPLIED,\r\n# INCLUDING BUT NOT LIMITED TO NON-INFRINGEMENT, MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE.\r\n# See LICENSE in the root of the software repository for the full text of the License.\r\n\r\nSPECIAL_MARK\r\n"
```

AI:

Now I can see that `SPECIAL_MARK` is at line 10, which matches the error message. The error says it expected "(" but got a newline, which suggests that `SPECIAL_MARK` replaced some CMake function call or command that should have parentheses.

Let me look at the project structure to understand what files are available and get more context about what should be in this CMakeLists.txt file.

• • •

AI:

Let me try one more approach. I'll use the exact format from the system prompt example but with the correct line numbers for this file.

Tool Invoke:

(git apply --check)

Tool Return:

"error: corrupt patch at line 7\r\n"

AI:

I think there might be an issue with the patch format itself. Let me try the simplest possible patch that just replaces the single line with two lines, using the exact format from the system prompt example.

Tool Invoke:

(git apply --check)

Tool Return:

"error: corrupt patch at line 7\r\n"

AI:

Based on my analysis, I understand what needs to be fixed. The `SPECIAL_MARK` in the CMakeLists.txt file replaced the CMake commands that should be there. Based on the working examples in the same project, the correct content should be:

1. `set_source_files_properties(basic_matmul.cpp PROPERTIES LANGUAGE ASCEND)`
2. `catlass_example_add_executable(00_basic_matmul_COMP dav-c220-cube basic_matmul.cpp)`

The patch should replace `SPECIAL_MARK` with these two lines. Let me provide the final answer in the correct format.

```
```answer_patch
diff --git b/CMakeLists.txt a/CMakeLists.txt
--- b/CMakeLists.txt
+++ a/CMakeLists.txt
@@ -9,1 +9,2 @@
-SPECIAL_MARK
+set_source_files_properties(basic_matmul.cpp PROPERTIES LANGUAGE ASCEND)
+catlass_example_add_executable(00_basic_matmul_COMP dav-c220-cube basic_matmul.cpp)
```
```

Figure S10: A backward task trajectory example of LangChain (DeepSeek-V3.1-terminus) (Answer).

```

User:
You are now operating within a docker container. container_name is `apiday--deepseek-ai--deepseek-v3.1-terminus--triton-ascend_20250925_71c94dc--ascend--examples--
generalization_cases_COMP--__main__test_argmin.ins2_patch`.

==== System
...
cmake: cmake version 3.22.1
gcc: gcc (Ubuntu 11.4.0-1ubuntu1~22.04.2) 11.4.0
g++: g++ (Ubuntu 11.4.0-1ubuntu1~22.04.2) 11.4.0
npu-smi info:
+-----+
| npu-smi 25.2.0          Version: 25.2.0          |
+-----+
project dir: /root/cann/LiveCANNBench/tasks/tasks_v2_1to3line/triton-ascend_20250925_71c94dc/triton-ascend_20250925_71c94dc--ascend--examples--
generalization_cases_COMP
SOC_VERSION: Ascend910B4
cann-toolkit: 8.2.rc1
...

==== Before Augmentation
Before augmentation, if running the below bash command
`bash
bash /root/cann/LiveCANNBench/tasks/tasks_v2_1to3line/triton-ascend_20250925_71c94dc/triton-ascend_20250925_71c94dc--ascend--examples--
generalization_cases_COMP/__main__test_argmin.sh 2>&1 | tee /root/cann/LiveCANNBench/tasks/tasks_v2_1to3line/triton-ascend_20250925_71c94dc/triton-
ascend_20250925_71c94dc--ascend--examples--generalization_cases_COMP/__main__test_argmin.log
`
the content printed in the terminal is:
`text
===== test session starts =====
platform linux -- Python 3.10.19, pytest-8.3.2, pluggy-1.6.0
rootdir: /root/cann/ascend_gitee_v2/triton-ascend_20250925_71c94dc
configfile: pyproject.toml
plugins: xdist-3.6.1
collected 1 item

ascend/examples/generalization_cases_COMP/test_argmin.py .           [100%]

===== warnings summary =====
ascend/examples/generalization_cases_COMP/test_argmin.py:219
/root/cann/ascend_gitee_v2/triton-ascend_20250925_71c94dc/ascend/examples/generalization_cases_COMP/test_argmin.py:219: PytestUnknownMarkWarning: Unknown
pytest.mark.shape_4d_5d - is this a typo? You can register custom marks to avoid this warning - for details, see https://docs.pytest.org/en/stable/how-to/mark.html
@pytest.mark.shape_4d_5d

ascend/examples/generalization_cases_COMP/test_argmin.py:287
/root/cann/ascend_gitee_v2/triton-ascend_20250925_71c94dc/ascend/examples/generalization_cases_COMP/test_argmin.py:287: PytestUnknownMarkWarning: Unknown
pytest.mark.shape_4d_5d - is this a typo? You can register custom marks to avoid this warning - for details, see https://docs.pytest.org/en/stable/how-to/mark.html
@pytest.mark.shape_4d_5d

-- Docs: https://docs.pytest.org/en/stable/how-to/capture-warnings.html
===== 1 passed, 2 warnings in 25.39s =====
...

==== During Augmentation
During augmenting, some lines in the following files have been replaced with a single `SPECIAL_MARK`:
test_common.py
...

==== After Augmentation
After augmentation, if running the below bash command
`bash
bash /root/cann/LiveCANNBench/tasks/tasks_v2_1to3line/triton-ascend_20250925_71c94dc/triton-ascend_20250925_71c94dc--ascend--examples--
generalization_cases_COMP/__main__test_argmin.sh 2>&1 | tee /root/cann/LiveCANNBench/tasks/tasks_v2_1to3line/triton-ascend_20250925_71c94dc/triton-
ascend_20250925_71c94dc--ascend--examples--generalization_cases_COMP/__main__test_argmin.ins2_errlog
`
the content printed in the terminal is:
`text
Traceback (most recent call last):
  File "/root/cann/ascend_gitee_v2/triton-ascend_20250925_71c94dc/ascend/examples/generalization_cases_COMP/test_argmin.py", line 12, in <module>
    import test_common
  File "/root/cann/ascend_gitee_v2/triton-ascend_20250925_71c94dc/ascend/examples/generalization_cases_COMP/test_common.py", line 160
    SPECIAL_MARK
IndentationError: expected an indented block after 'elif' statement on line 159
[ERROR] 2026-01-02 08:44:19 (PID:1085, Device:-1, RankID:-1) ERR99999 UNKNOWN applicaiton exception
[TaskError] Script failed. Return code: 1
...

==== Task
Give me a patch that can fix the project (i.e., replace `SPECIAL_MARK` with the correct content).
Such that the execution results of the fixed project are as close as possible to those of the original project before augmentation.
Do not modify any of the project files.
Save your patch as the patch file `/root/cann/ascend_gitee_v2/triton-ascend_20250925_71c94dc/ascend/examples/generalization_cases_COMP/__answer__.patch`.
I will later re-initialize a new container for the project and verify your patch there, i.e.,
`bash
cd /root/cann/ascend_gitee_v2/triton-ascend_20250925_71c94dc/ascend/examples/generalization_cases_COMP && git apply /root/cann/ascend_gitee_v2/triton-
ascend_20250925_71c94dc/ascend/examples/generalization_cases_COMP/__answer__.patch
`
Therefore, do not modify any of the project files, which will cause inconsistency during `git apply`.
The absolute-git dir of `/root/cann/ascend_gitee_v2/triton-ascend_20250925_71c94dc/ascend/examples/generalization_cases_COMP` is `/root/cann/ascend_gitee_v2/triton-
ascend_20250925_71c94dc/ascend/examples/generalization_cases_COMP.git`.

Below is an example patch from other projects:
`example_patch
diff --git b/main.cpp a/main.cpp
--- b/main.cpp
+++ a/main.cpp
@@ -20,7 +20,7 @@ int32_t main(int argc, char const *argv[])

// 8 indicates the kernel function will be executed on eight cores.
constexpr uint32_t blockDim = 8;
-SPECIAL_MARK
+ hello_world_do(blockDim, stream);
aclrtSynchronizeStream(stream);
...
aclrtDestroyStream(stream);
`

```

Figure S11: A backward task trajectory example of mini-SWE-Agent (DeepSeek-V3.1-terminus) (Prompt).

AI:

THOUGHT: The error message indicates an `IndentationError` in `test_common.py` at line 160, where `SPECIAL_MARK` is placed after an `elif` statement without proper indentation. I need to examine the context around line 159-160 in `test_common.py` to understand what the original code might have been. The goal is to replace `SPECIAL_MARK` with the correctly indented code block that was originally there. Since the original test passed with warnings, the fix should restore the original logic. I'll first look at the file to understand the structure.

```
```bash
nl -ba /root/cann/ascend_gitee_v2/triton-
ascend_20250925_71c94dc/ascend/examples/generalization_cases_COMP/test_common.py | sed -n '150,170p'
```
```

Tool Invoke:

(terminal)

Tool Return:

```
"<returncode>0</returncode>\n<output>\n 150\t elif dtype == 'bool':\n 151\t return torch.randint(low=0,\nhigh=2, size=shape).bool()\n 152\t else:\n 153\t raise ValueError('Invalid parameter \\\"dtype\\\" is\nfound: {}'.format(dtype))\n 154\t\n 155\t\n 156\tdef get_triton_sig_tynename(dtype):\n 157\t if dtype ==\n'float32':\n 158\t tyname = \"*fp32\"\n 159\t elif dtype ==\n'int32':\n 160\tSPECIAL_MARK\n 161\t tyname = \"*i64\"\n 162\t elif dtype ==\n'float16':\n 163\t tyname = \"*fp16\"\n 164\t elif dtype == 'int16':\n 165\t tyname =\n\"*i16\"\n 166\t elif dtype == 'int8':\n 167\t tyname = \"*i8\"\n 168\t elif dtype ==\n'bool':\n 169\t tyname = \"*i1\"\n 170\t else:\n</output>"
```

• • •

Tool Invoke:

(terminal)

Tool Return:

```
"<returncode>0</returncode>\n<output>\nSyntax is valid\n</output>"
```

AI:

THOUGHT: The syntax validation passed, which confirms that the patch has fixed the indentation error. Now I should test the original script to make sure it runs without the syntax error that was occurring before.

```
```bash
cd /root/cann/ascend_gitee_v2/triton-ascend_20250925_71c94dc/ascend/examples/generalization_cases_COMP && python
test_argmin.py 2>&1 | head -20
```
```

Tool Invoke:

(terminal)

Tool Return:

```
"<returncode>0</returncode>\n<output>\nTraceback (most recent call last):\n File \"/root/cann/ascend_gitee_v2/triton-
ascend_20250925_71c94dc/ascend/examples/generalization_cases_COMP/test_argmin.py", line 5, in <module>\n import
pytest\nModuleNotFoundError: No module named 'pytest'\n</output>"
```

AI:

THOUGHT: The script is now running without the indentation error, which means our patch has successfully fixed the syntax issue. The `ModuleNotFoundError` for `pytest` is expected in this minimal environment and is not related to our fix. The original error was an `IndentationError` which has been resolved. Our patch is correct and ready for submission.

```
```bash
echo COMPLETE_TASK_AND_SUBMIT_FINAL_OUTPUT
```
```

Figure S12: A backward task trajectory example of mini-SWE-Agent (DeepSeek-V3.1-terminus) (Answer).