BUILD-BENCH: Benchmarking LLM Agents on Compiling Real-World Open-Source Software

Zehua Zhang, Ati Priya Bajaj, Divij Handa, Siyu Liu, Arvind S Raj, Hongkai Chen,

Hulin Wang, Yibo Liu, Zion Leonahenahe Basque, Souradip Nath, Vishal Juneja,

Nikhil Chapre, Yan Shoshitaishvili, Adam Doupé, Chitta Baral, Ruoyu Wang
School of Computing and Augmented Intelligence
Arizona State University
Tempe, AZ 85281, USA
{zzhan645, abajaj7, dhanda, chitta, fishw}@asu.edu

Abstract

Automatically compiling open-source software (OSS) projects is a vital, labor-intensive, and complex task, which makes it a good challenge for LLM Agents. Existing methods rely on manually curated rules and workflows, which cannot adapt to OSS that requires customized configuration or environment setup. Recent attempts using Large Language Models (LLMs) used selective evaluation on a subset of highly rated OSS, a practice that underestimates the realistic challenges of OSS compilation. In practice, compilation instructions are often absent, dependencies are undocumented, and successful builds may even require patching source files or modifying build scripts. We propose a more challenging and realistic benchmark, BUILD-BENCH, comprising OSS that are more diverse in quality, scale, and characteristics. Furthermore, we propose a strong baseline LLM-based agent, OSS-BUILD-AGENT, an effective system with enhanced build instruction retrieval module that achieves state-of-the-art performance on BUILD-BENCH and is adaptable to heterogeneous OSS characteristics. We also provide detailed analysis regarding different compilation method design choices and their influence to the whole task, offering insights to guide future advances. We believe performance on BUILD-BENCH can faithfully reflect an agent's ability to tackle compilation as a complex software engineering tasks, and, as such, our benchmark will spur innovation with a significant impact on downstream applications in the fields of software development and software security.

1 Introduction

Imagine that you are a graduate student during a rebuttal period. The reviewers strongly suggested that you compare your system with prior work. It was only published a few years ago, so you find the GitHub repo, download the code, and read the included docs. It doesn't compile. The dependency URLs are missing. Required libraries aren't included. Even if it worked perfectly when first published, it's going to take you days to even compile this system. This scenario highlights the difficulty of compiling once-maintained open-source code, and in fact this problem is faced by the broader software engineering community. However, recent advances in Large Language Models (LLMs) promise to improve various software engineering tasks [Brown et al., 2020, Touvron et al., 2023, Chen et al., 2021]. While commercial software can be developed with stringent and consistent engineering practices, OSS projects are highly heterogeneous. Additionally, OSS projects are maintained by varied

39th Conference on Neural Information Processing Systems (NeurIPS 2025).

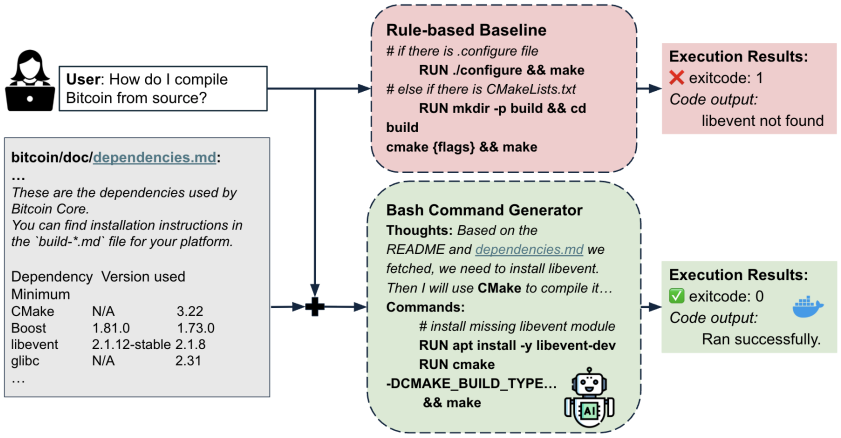

Figure 1: Demonstration of rule-based and AI agentic compilation methods. While rule-based methods follow a predefined workflow, they cannot adequately adapt to different environments. In comparison, AI agents leverage their pre-trained knowledge to adjust the compilation commands based on execution results. In this example, the agent realizes LIBEVENT is a key missing dependency for Bitcoin to compile and installs it to successfully compile the project.

contributors, adopt various build frameworks, and frequently require platform-specific configurations.

Compiling OSS often requires manual intervention to resolve missing dependencies, version mismatches, or undocumented environment requirements. Although prior rule-based methods (e.g., GHCC [Hu, 2020] and Assemblage [Liu et al., 2024]) attempted to automate this process by iteratively invoking build scripts, they cannot robustly handle dependency, toolchain, or platform mismatches. These challenges impact human software engineers who integrate OSS into their own applications, as this requires compilation to turn the software into a library or binary that they can use in their own system. Reliable and scalable automated compilation, in addition to improving software engineering, has other research benefits: It enables large-scale usage of binary data sources, supports program analysis and vulnerability discovery, and accelerates software maintenance workflows [Lacomis et al., 2019, Dramko et al., 2023, Pal et al., 2024]. This work addresses these challenges by using LLM-based agents to automate OSS compilation at scale.

LLMs that are pre-trained on a large amount of natural language data exhibit strong performance in general-purpose tasks, even with zero-shot prompting [Kojima et al., 2023]. This capability is generalized to software engineering-related tasks, such as code generation, software debugging, documentation generation, and code refactoring. As such, LLM-based AI agents [Yao et al., 2023, Shinn et al., 2023], which are autonomous systems that use LLMs to iteratively plan, reason, and act, are increasingly used to automate and facilitate complex software engineering workflows [Wang et al., 2024]. In this context, we position OSS compilation as a challenging and underexplored target for LLM-based agents. We are thus motivated to create a benchmark (**BUILD-BENCH**) and systematically evaluate various, specifically agentic, solutions. BUILD-BENCH includes 148 humanly verified repositories out of 385 randomly selected C/C++-heavy OSS from GitHub, each manually annotated for compilation success and build instruction retrieval. We use an additional 70 carefully chosen projects as a validation set to support the development of our agentic baseline method, **OSS-BUILD-AGENT**. Using BUILD-BENCH, we evaluate existing rule-based and LLM baselines and agentic compilation methods. We showcase the current shortcoming of rule-based methods in compilation performance and success verification. For LLM and agentic compilation methods, we inspect in detail the performance discrepancy of various compilation system designs. Specifically, we demonstrate the effectiveness of our LLM-Assisted Retrieval and Multi-Agent Compilation module design through a side-by-side comparison to another agentic solution. Through the release of BUILD-BENCH and our analysis, we encourage researchers to create better agentic solutions for the compilation task, which will ultimately benefit the AI, software engineering, and software security communities.

Contributions. Our contributions are as follows:

- We created **BUILD-BENCH**, which is a benchmark that contains both a hand-picked validation set and a randomly sampled test set with manual inspection and labeling to support a rigorous and systematic evaluation of different OSS compilation techniques.

- We evaluated the performance of five rule-based and AI build methods on BUILD-BENCH, including two agentic methods. Our proposed OSS-BUILD-AGENT achieved the best performance, surpassing the strongest rule-based baseline by roughly 50%. With a strong base LLM, OSS-BUILD-AGENT reached a 66.4% success rate, establishing a strong baseline performance, while BUILD-BENCH remains a challenge for future research.

- We offered a detailed inspection of various design approaches in compilation instruction retrieval and error resolution modules and their effects on task performance.

2 Constructing BUILD-BENCH

A benchmark for automated OSS compilation requires a representative dataset of OSS. We first analyze the prior work COMPILEAGENTBENCH [Hu et al., 2025], which also targets the automatic compilation task. Specifically, it consists of 100 popular and well-known GitHub projects, averaging over 8,000 stars. However, this focus on popular repositories overlooks the vast majority of OSS: 99.88% of C/C++ projects on GitHub have fewer than 500 stars. Therefore, the generalizability of evaluation results on COMPILEAGENTBENCH may be undermined by projects that are unusually well documented, well maintained, and less representative of the in-the-wild challenges.

Data Filtering. Therefore, we strive to create BUILD-BENCH as a statistically representative benchmark to ensure generalizability to the broad diversity of OSS. To create the raw dataset, we collected 2.77M C and 4.23M C++ repositories from GitHub RESTAPI with a date range between April 1st, 2008 to January 1st, 2024. To remove extremely low-quality repositories, we apply a few filters: We exclude projects with names or descriptions that contained certain keywords (e.g., `homework` or `assignment`, more in Appendix A) or have a stargazer count below 50 to ensure the OSS are meaningful for both practical usage and research purposes. For deduplication, we excluded repositories that are forks of other repositories. After filtering, the raw dataset contains 6.57M repositories. From this population, we randomly select 385 projects, the minimum sample size required to measure a population proportion with 95% confidence with a margin of error of 5%, according to Cochran [1977] (details in Appendix B). We believe that this randomly sampling helps ensure that BUILD-BENCH better approximates real-world OSS compilation challenges.

Data Selection and Labeling by Human Experts. Due to the random sampling process, we cannot guarantee that all of the 385 projects can be compiled. Therefore, human experts manually built each repository to determine its validity. We also exclude OSS repositories that fit into following criteria: (a) The repository targets another operating system and cannot be cross-compiled; (b) The repository only contains trivial or unbuildable content; (c) The repository is missing critical source files and broken dependencies that cannot be installed or created; (d) There are compilation and linking errors that human experts cannot resolve in a best-effort setting.

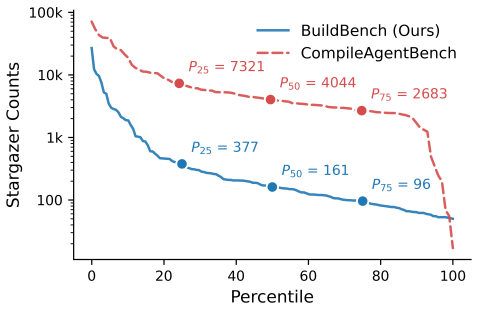

Figure 2: Distribution of stargazer counts of BUILD-BENCH and COMPILEAGENTBENCH. Note that in BUILD-BENCH, relatively low-profile repositories made up the majority of the samples.

This process resulted in 148 compilable repositories as the final test set. Additionally, we manually created ground truth labels for compiled binary file names and URLs where the build instructions are hosted, if provided by the developers. The manual labeling involved 12 graduate students with more than 3 years of experience working on system research, and took roughly 150 hours.

We also created a validation set of 70 popular repositories used to evaluate OSS-BUILD-AGENT.

The representativeness (or diversity) of a benchmark for compilation task is essential to evaluate the generalizability and performance of any compilation technique. We analyze the representativeness from two following aspects: popularity distribution and build system distribution.

Popularity. The number of Stargazers (or stars) of a GitHub repository is often used to approximate popularity and perceived quality [Dramko et al., 2023]. A higher number often correlates with popular or essential functionality, better code quality, and an active development community that supports continuous and frequent maintenance. However, a majority of repositories tend to have relatively lower stars compared to high-profile projects such as OpenSSL or FFmpeg. This is partially because repositories are often created for specific use cases and targeting smaller and specialized audience groups instead of having widely applicable use cases. Meanwhile, most repositories are for personal or experimental use, further undermining their limited visibility.

Figure 2 shows the Stargazer counts of repositories in BUILD-BENCH and COMPILEAGENTBENCH. Most repositories in BUILD-BENCH are in the 50–500 range, indicating random selection results coincide with the long-tail distribution of overall repository popularity. This characteristic makes BUILD-BENCH more challenging for evaluating compilation techniques, because low-profile repositories often lack documentation or require additional customization or configuration. In contrast, the Stargazer counts of COMPILEAGENTBENCH repositories are aggregated between 2k and 10k, and these popular repositories might be considered as an underestimation of the true difficulty of the compilation task.

Build Systems. We further analyze the build systems and tool chains used in BUILD-BENCH repositories: 62 projects use `Make`, 60 use `CMake`, 29 use `Autotools`, and 14 use Visual Studio (`MSBuild`). Smaller—but non-negligible subsets—adopt alternative systems such as custom scripts, `QMake`, `Meson`, etc. 10 repositories provide no explicit build system, often relying on direct compilation. This diversity showcases the heterogeneity of real-world OSS, where the build system selection often depends on the project domain, platform, and community preference. Note that there may be multiple build systems available in the same OSS, which offers alternative compilation approaches.

Overall, the results show that BUILD-BENCH adequately represents a wide variety of real-world C and C++ projects and is suitable as a benchmark for evaluating of automated build techniques.

3 Agentic Building Methods

We create an agentic compilation technique, OSS-BUILD-AGENT. As Figure 3 shows, an initial (and optional) LLM iteratively extends the README with additional compilation instructions, then a multi-agent build system iteratively generates and executes compilation steps.

3.1 Compilation Instruction Retrieval

Many repositories with complex build processes or that require specialized configurations tend to document these steps for human developers. Accessing these instructions is crucial because they provide agents with helpful information regarding necessary setup and configuration steps.

However, we find that this documentation is not only located in the OSS repository's README, but can also be located in other files in the repository or on another website. To solve this challenge we propose an LLM-Assisted Retrieval module, an optional component that precedes OSS-BUILD-AGENT. Our approach uses an out-of-box LLM as an incremental retriever to synthesize the complete set of instructions required for compiling a given repository.

The process uses the project's README as input. The LLM then iteratively performs three operations: (i) it distills potential compilation instructions from the file, (ii) it evaluates the sufficiency of the acquired information, and (iii) if the information is not sufficient enough to support compilation, it identifies promising links, encompassing both internal files and external web pages. The contents of up to three newly identified links are subsequently fetched, summarized, and re-evaluated. This recursive process of retrieval and refinement continues until the LLM's confidence in the completeness of the build knowledge is fulfilled or a maximum of three iterations is reached. The output of this is the final compilation instructions.

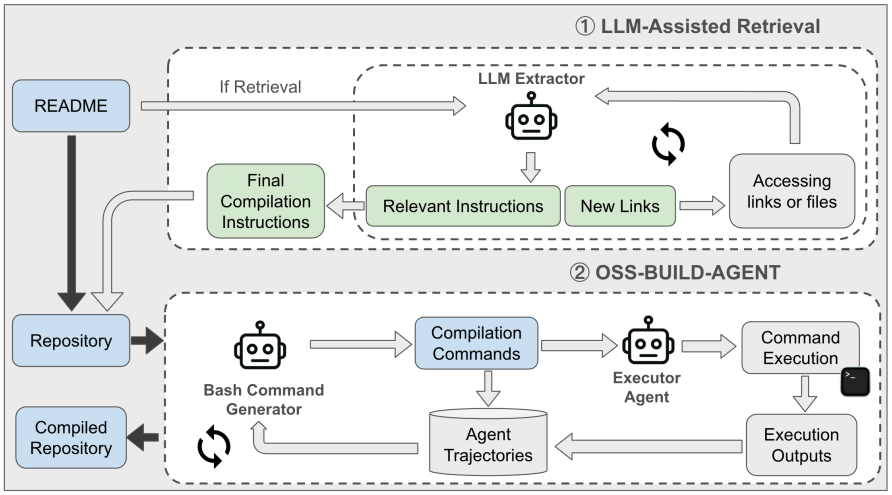

Figure 3: OSS-BUILD-AGENT system diagram. The initial input is the README, then an optional LLM extends this with additional compilation instructions. Finally, a multi-agent build system iteratively generates and executes compilation steps, attempting to compile the target repository.

3.2 Multi-Agent Compilation System

The compilation system comprises two cooperating agents, both using an out-of-the-box LLM of the user's selection: *Bash Command Generator* is given the final compilation instructions from the prior module (if using LLM-Assisted Retrieval) and the repository as input and produces a candidate sequence of bash commands to compile the repository. *Execution Agent* runs these commands within a containerized environment and returns the execution results. Prompts are included in Appendix Section C.2 and C.3.

For refinement steps $k = 0, \ldots, K$, let C_k be the input into *Bash Command Generator*, which produces S_k, the commands to be executed by *Execution Agent* in the environment that returns execution results f_k.

Initialization. *Bash Command Generator* produces the first set of commands directly from the input prompt C_0 because there is no execution feedback. *Execution Agent* runs generated commands S_0 in a fresh Docker container, yielding the initial execution results f_0.

Iterative Error Resolution. This constitutes the standard agentic loop: *Bash Command Generator* uses both C_k and the latest execution results f_k to craft revised commands S_k, which the *Execution Agent* then executes again.

The process ends when $\text{Success}(f_k) = \text{true}$ or when $k = K$, exceeding the maximum turns allowed. This iterative error resolution process enables the compilation to recover from missing dependencies, incorrect flags, or environmental mismatches, an ability that is required for OSS compilation task, as we analyzed in Section 6.

4 Baseline Methods

In this section, we present existing rule-based techniques as well as two LLM-based compilation methods we compare against.

GHCC. GHCC [Hu, 2020] is a rule-based tool for building GitHub repositories. Prior research uses datasets that GHCC created [Lacomis et al., 2019, Xie et al., 2024]. Given a repository, GHCC attempts to build the project by first discovering all build system-specific files (e.g., Makefile and CMakeLists.txt) and then conducting a rule-based build routine customized for these build systems.

Assemblage. Assemblage [Liu et al., 2024] is a system designed to automate the construction of binary datasets of primarily Windows executables by building source code. It follows a similar rule-based compilation workflow as GHCC.

Single-turn LLM baseline. To evaluate the project-building performance of pretrained LLMs on a single-turn basis, we prompt an out-of-the-box LLM to generate a set of Bash commands to build a target repository and execute the commands in a Docker container. The input to this baseline is the README file and file directory of the OSS's root directory. Without any execution feedback, this single-turn baseline cannot adjust its initial output. (Prompt in Appendix C.1.)

CompileAgent. CompileAgent [Hu et al., 2025] also introduces a multi-agentic compilation system. It adopts a flow-based agent strategy in which a master agent orchestrates the build process across two core modules: (1) CompileNavigator for locating and extracting build instructions and (2) ErrorSolver for resolving compilation errors. These modules are supported by five specialized tools (shell execution, file navigation, instruction extraction, web search, and multi-agent discussion), four of which involve auxiliary LLM agents, totaling seven agents in the pipeline. We include its official open-source implementation as a baseline in our evaluation to provide a representative comparison against our agentic approaches.

5 Experiment Setup and Evaluation Methods

We evaluate the performance of baseline build techniques and OSS-BUILD-AGENT on BUILD-BENCH. We implement single-turn LLM baseline with two base models: GPT o3-mini and Claude 3.7-Sonnet. For OSS-BUILD-AGENT, we use five models, representing diverse characteristics including reasoning vs. non-reasoning, generic vs. coding-specific, and different parameter sizes. For CompileAgent we use GPT-4o as the main base model as its implementation indicates.

All build methods build each repository in a fresh Ubuntu 22.04 Docker container, with minimal packages pre-installed.

Success Metrics. A key evaluation challenge is to determine if a build method successfully builds a given repository. Existing build methods determine the compilation process as **Completion** with the presence of at least one binary post-building. This metric is unreliable when (1) a failed building process generates intermediate binary files, or (2) a submodule (or a vendored package) successfully builds while the main repository fails building.

We improve the completion success criteria with additional validation using expert-generated, per-repository lists of binary file names as ground truth. After the building of a repository completes, we compare the file names of all produced binary files against a expert-generated list. We categorize success into two types: (1) **Strict Success** only when all binary file names in the expert-generated list exist, and (2) **Flexible Success** when at least one file name in our expert-generated list exists.

6 Evaluating Build Methods

Table 1 presents the performance of all build methods on BUILD-BENCH.

Baselines. For rule-based methods, GHCC achieves 30.2% completions and 13.4% flexible validated successes, outperforming Assemblage. Single-turn LLM baselines' results vary: o3-mini exhibits degraded performance, while Claude 3.7-Sonnet is surprisingly strong for a non-agent setting (21.5% strict; 22.1% flexible). Moreover, the performance of COMPILEAGENT suffers a substantial drop from 89% strict validated success on COMPILEAGENTBENCH to 49.7% strict and 55.7% flexible on BUILD-BENCH. This performance drop indicates a pronounced distribution shift and higher difficulty of BUILD-BENCH.

Agents enable compilation error resolution in multi-turn setting. OSS-BUILD-AGENT substantially outperforms all rule-based baselines. The best configuration, OSS-BUILD-AGENT with LLM-assisted Retrieval using Claude 3.7-Sonnet, reaches 66.4% strict and 71.8% flexible validated successes, a gain of 49.7 percentage points over single-turn baseline with the same model. Iterative observation–repair–rebuild loops allow agents to access and receive feedback from execution results, backtrack from ineffective commands, and apply targeted fixes that single-turn approaches cannot.

Agentic build methods are model-agnostic, but scale with model intelligence. The agent framework uses out-of-the-box pre-trained LLMs, allowing our framework to be model-agnostic. Nevertheless, performance scales with model capability. Among all settings, Claude 3.7-Sonnet achieves the best performance with significant margin to the next best model. This confirms that stronger

Table 1: Performance of all evaluated build techniques on BUILD-BENCH test set. Section 5 describes the evaluation metrics of completion and validated successes.

LLM Usage	Build Method	Un-validated Completions %	Validated Successes %	
			Strict	*Flexible*
N/A	GHCC	30.2	10.1	13.4
N/A	Assemblage	10.7	6.0	9.4
Single Turn	LLM baseline (o3-mini)	9.4	7.4	8.1
Single Turn	LLM baseline (Claude 3.7-Sonnet)	23.5	21.5	22.1
Multi-Agents	CompileAgent (GPT-4o with Retrieval)	N/A	49.7	55.7
Multi-Agents	**OSS-BUILD-AGENT *w/o Retrieval (Ours)***			
	GPT-4o	56.8	38.5	41.9
	GPT o3-mini	67.6	48.0	50.7
Multi-Agents	**OSS-BUILD-AGENT *w/ LLM-Assisted Retrieval (Ours)***			
	GPT-4o (Avg of 3 Runs)	70.2	53.0	57.6
	GPT o3-mini	79.9	63.1	68.5
	Claude 3.7-Sonnet	**85.2**	**66.4**	**71.8**
	Gemini-2.5-flash	77.2	57.0	61.1
	Qwen3 235B	83.9	59.7	66.4
	Qwen3 Coder 485B	48.3	34.2	38.9

LLMs are more effective in adjusting its output based on error results and applying targeted fixes, two skills that are central to resolving complex build failures. In contrast, smaller models (e.g., o3-mini) perform consistently but saturate at around 68–69% flexible success, while specialized models (Qwen3 Coder) underperform (38.9% flexible), suggesting that coding specialization may become a drawback, considering retrieval module challenges more on model's documentation comprehension ability. Overall, the performance of OSS-BUILD-AGENT is model-agnostic, but stronger LLMs still improve the performance of OSS-BUILD-AGENT.

6.1 Instability and Repeated Experiments

Table 2: Results from three repeated runs of OSS-BUILD-AGENT with retrieval using GPT-4o. k refers to the order in Figure 4. Error Fixing attempts is the average of attempts across all repositories in one run.

k	Error Fixing Attempts	Strict Success %	Flexible Success %
$k = 2$	4.8	45.6	50.3
$k = 1$	6.9	54.7	59.5
$k = 3$	8.4	58.8	62.9

Instability in agentic frameworks is a well-recognized issue [Yao et al., 2024]. Although OSS-BUILD-AGENT performs strongly, its results fluctuate over runs. To quantify this, we repeat experiments with GPT-4o, a non-reasoning model, as the base model in three independent runs. Table 2 shows the results, where OSS-BUILD-AGENT achieves 53.0% ± 6.8 strict and 57.6% ± 6.5 flexible validated success, indicating non-trivial variance. We attribute this to two major factors. First, LLM-guided retrieval can follow different documentation accessing trajectories and produce different build recipes across runs, shifting the subsequent compilation trajectories. Second, LLM outputs are non-deterministic even with identical prompts [Song et al., 2024], and this randomness compounds over multi-turn interactions. Together, these effects lead to instability.

Additionally, we evaluate pass@k across three runs to assess the benefit of multiple attempts (Figure 4). For the strict setting, the pass rates increase from 54.7% at pass@1 to 59.5% at pass@2 and 65.5% at pass@3. Under the flexible setting, the corresponding rates are higher, rising from 59.5% to 64.2% and 70.3%. These results demonstrate that multiple agentic trials substantially improve performance, which may better control the stochastic nature of AI agents. Repeated experiments not only control for performance variance, but also help to validate the arguments based on performance.

6.2 Retrieval and Error Resolution

Despite the architectural differences between COMPILEAGENT and OSS-BUILD-AGENT, they both incorporate two similar modules of build instruction retrieval and agentic error resolution. We discuss the system design differences and their performance impact.

Retrieval Analysis. Accurate retrieval of build instructions has a strong impact on subsequent compilation performance. Developer-provided instructions offer a solid starting point that agents can adapt to match specific configuration requirements or environment differences. In BUILD-BENCH, we identified 130 OSS repositories from BUILD-BENCH test set with clear URL labels for the build instruction. Together, these form a secondary benchmark for evaluating the retrieval module described in Section 3.1.

We evaluate COMPILEAGENT on the 130 OSS repositories with URL labels along with OSS-BUILD-AGENT's LLM-Assisted Retrieval (both using GPT-4o), using the same criteria for success: whether the retrieval module accessed the ground-truth URL that hosts the build instruction for the given repository. In our evaluation, the retrieval module of OSS-BUILD-AGENT achieved a retrieval accuracy of 73.8%, significantly outperforming COMPILEAGENT's 46.2%. We attribute this performance improvement to key design choices in our retrieval module.

We observe that COMPILEAGENT's retrieval tool favors certain files or pages and often avoids less obvious links, leading to missed instructions. For example, when given the structure of the root directory of a repository, agents usually pick build scripts (e.g., `Makefile`) as the retrieval target. Unfortunately, build scripts are often too noisy and can divert the agents from continuing to find explicit documentation about configuration or setup. Additionally, build instructions can exist across multiple sources (e.g., README files, wiki pages, and subdirectories), and the derailment of agents compounds when facing noises from the scattered instructions. In comparison, we design the LLM-Assisted Retrieval module of OSS-BUILD-AGENT as a workflow that mimics a human engineer. It focuses on exploring the documentation instead of the build process. In the first iteration of retrieval, we instruct the LLM to inspect the main README file to extract information or find useful URLs. This prevents the LLM from being distracted by build scripts. Traversing a path of documentation files, our retrieval module better handles scattered information.

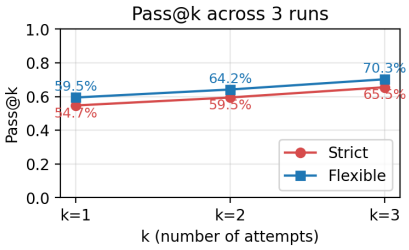

Figure 4: pass@k performance of OSS-BUILD-AGENT with LLM-Assisted Retrieval using GPT-4o. For $K = 1$, we report the earliest chronological run.

Error Resolution Attempts. We compare two different agentic systems and manually inspect their action trajectories of error resolution. We observe that while COMPILEAGENT employs a variety of tools, the main agent rarely invokes some of these tools (such as Multi-Agent Discussion for error resolution). The master agent usually exits too "easily" when encountering compilation errors, without attempting more fixes by invoking tools. Because compilation errors are often long, verbose, and nested, locating and fixing root-cause errors may require iterative attempts (interested readers may refer to an example in Appendix F). Thus, more error resolving attempts is favorable, which is validated by our repeated experiments with OSS-BUILD-AGENT as Table 2 shows. Using the same base model, we observe that validated success rate scales well with the number of attempts to resolve the error.

Despite the scaling effect of error resolution attempts, OSS-BUILD-AGENT attempts 6.6 times on average, in comparison to 7.5 times in COMPILEAGENT (excluding its retrieval module for fair comparison). This difference is due to our agent outputting the entire set of build commands, while COMPILEAGENT outputs one bash command at a time and refines it iteratively if execution shows error. While this fine-grained approach can be effective, it also inflates the trajectory with trivial commands (e.g., ls, mkdir) that rarely fail but still count as separate steps. Conversely, OSS-BUILD-AGENT generates a more complete set of compilation commands intended to drive the build to completion in a single run, followed by troubleshooting if needed. This design allows the agent to observe the full command history at each step, providing contextual information for error resolution. For example (details in Appendix E), an error such as *The source directory does not appear to*

contain CMakeLists.txt can be resolved more effectively when the agent has access to prior directory navigation steps, enabling it to adjust the working directory and retry seamlessly.

Together, our agentic designs in OSS-BUILD-AGENT ensure better retrieval and error resolution practices achieve better performance despite using only two agents (vs. seven in COMPILEAGENT) and simpler architectural design, showcasing the competitive performance of our end-to-end agentic compilation pipeline.

6.3 Failure Modes of Agentic Build Methods

Agentic methods are known for instability, task derailment, disobeying instructions, and many other drawbacks [Cemri et al., 2025]. Thus, it is important to identify the failure modes of agents to facilitate future development of more potent agents for our task. We manually inspect the building process executed by the agentic build method, which is built on the GPT o3-mini model and enhanced with the LLM-assisted retrieval approach. The results are shown in Figure 5. It includes errors in both the retrieval and compilation stages. The most common failure mode happens to 69 repositories, the agents could recognize the error messages but failed to produce fixes to eliminate the errors after many turns and decided to terminate. As dependency errors are often straightforward to solve but will lead to build failures if not fixed, we also identified the failure cases resulting from dependency errors.

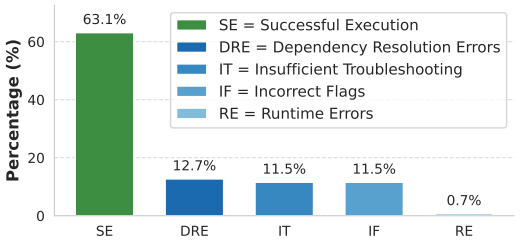

Figure 5: Agent failure modes analysis of OSS-BUILD-AGENT enhanced with LLM-Assisted Retrieval implemented with o3-mini.

7 Limitations

One of potential drawbacks of our benchmark is the relatively small number of compilable repositories for the test set. However, we compensate for the quantity with intensive manual verification that produces ground-truth binary file and retrieval labels that facilitate both analysis and future developments.

Although OSS-BUILD-AGENT shows competitive performance, we acknowledge the inherent instability of the agentic framework that may introduce variations in performance when reproducing the experiments. Also, we believe that agentic retrieval could be further enhanced with recent advancement of AI agent research, which ultimately improves the accuracy of the retrieval to enhance the overall compilation performance. We also invite researchers to expand the potential of different agents' design philosophies and validate them on BUILD-BENCH to facilitate real-life developers and downstream research.

8 Related Work

LLM has shown promising performance across various software engineering tasks. These include automated resolution of GitHub issues [Jimenez et al., 2024, Su et al., 2025], intelligent code generation [Ishibashi and Nishimura, 2024], automated test case generation [Pizzorno and Berger, 2025, Yuan et al., 2024], and Python software installation [Milliken et al., 2024]. Within this growing landscape, the task of automatically compiling C/C++ OSS remains relatively underexplored. The intricacies of these languages, including discontinued maintenance, complex build systems that depend on many external dependencies, and the often less informative error messages from compilers like GCC and Clang [Onyango and Mariga, 2023], all add up to the difficulties of the task. Rule-based methods have been used extensively in previous work on building binary datasets for downstream tasks [Hu, 2020, Lacomis et al., 2019, Liu et al., 2024]. While such methods suffer from their inherent fragility, AI agents may be a suitable solution. As initial efforts, such as CompileAgent [Hu et al., 2025], have indicated the potential of using the agentic compilation method, the dataset against which it is evaluated consists of many well-known OSS that may have their compilation processes memorized by LLM, introducing biases to the evaluation results. We believe it is necessary to create

a more challenging and representative benchmark dataset that allows for more insightful evaluation and analysis of agentic compilation methods.

9 Conclusion

In this paper, we present the first comprehensive benchmark for evaluating automated methods for building C and C++ source code repositories to construct binary datasets. We first create BuildBench, a representative dataset of 385 C and C++ code repositories. We systematically evaluate existing and newly proposed AI-based agentic build methods on BuildBench. Our evaluation provides valuable insights in agentic build methods.

References

Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language Models are Few-Shot Learners, July 2020. URL http://arxiv.org/abs/2005.14165. arXiv:2005.14165 [cs].

Mert Cemri, Melissa Z. Pan, Shuyi Yang, Lakshya A. Agrawal, Bhavya Chopra, Rishabh Tiwari, Kurt Keutzer, Aditya Parameswaran, Dan Klein, Kannan Ramchandran, Matei Zaharia, Joseph E. Gonzalez, and Ion Stoica. Why Do Multi-Agent LLM Systems Fail?, April 2025. URL http://arxiv.org/abs/2503.13657. arXiv:2503.13657 [cs].

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé, Jared Kaplan, Harrison Edwards, Yura Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mo Bavarian, Clemens Winter, Philippe Tillet, F. Such, D. Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William H. Guss, Alex Nichol, Igor Babuschkin, S. Balaji, Shantanu Jain, A. Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, M. Knight, Miles Brundage, Mira Murati, Katie Mayer, P. Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, I. Sutskever, and Wojciech Zaremba. Evaluating Large Language Models Trained on Code. *ArXiv*, July 2021. URL https://www.semanticscholar.org/paper/Evaluating-Large-Language-Models-Trained-on-Code-Chen-Tworek/acbdbf49f9bc3f151b93d9ca9a06009f4f6eb269.

William Gemmell Cochran. *Sampling Techniques*. Wiley, 1977. ISBN 978-81-265-1524-0. Google-Books-ID: xbNn41DUrNwC.

Luke Dramko, Jeremy Lacomis, Pengcheng Yin, Ed Schwartz, Miltiadis Allamanis, Graham Neubig, Bogdan Vasilescu, and Claire Le Goues. DIRE and its Data: Neural Decompiled Variable Renamings with Respect to Software Class. *ACM Trans. Softw. Eng. Methodol.*, 32(2):39:1–39:34, March 2023. ISSN 1049-331X. doi: 10.1145/3546946. URL https://dl.acm.org/doi/10.1145/3546946.

Li Hu, Guoqiang Chen, Xiuwei Shang, Shaoyin Cheng, Benlong Wu, Gangyang Li, Xu Zhu, Weiming Zhang, and Nenghai Yu. CompileAgent: Automated Real-World Repo-Level Compilation with Tool-Integrated LLM-based Agent System, May 2025. URL http://arxiv.org/abs/2505.04254. arXiv:2505.04254 [cs].

Zecong Hu. huzecong/ghcc: GitHub Cloner & Compiler, January 2020. URL https://github.com/huzecong/ghcc/tree/master.

Yoichi Ishibashi and Yoshimasa Nishimura. Self-Organized Agents: A LLM Multi-Agent Framework toward Ultra Large-Scale Code Generation and Optimization, April 2024. URL http://arxiv.org/abs/2404.02183. arXiv:2404.02183 [cs].

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. SWE-bench: Can Language Models Resolve Real-World GitHub Issues?, November 2024. URL http://arxiv.org/abs/2310.06770. arXiv:2310.06770 [cs].

Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. Large Language Models are Zero-Shot Reasoners, January 2023. URL http://arxiv.org/abs/2205.11916. arXiv:2205.11916 [cs].

Jeremy Lacomis, Pengcheng Yin, Edward J. Schwartz, Miltiadis Allamanis, Claire Le Goues, Graham Neubig, and Bogdan Vasilescu. DIRE: A Neural Approach to Decompiled Identifier Naming, October 2019. URL http://arxiv.org/abs/1909.09029. arXiv:1909.09029 [cs].

Chang Liu, Rebecca Saul, Yihao Sun, Edward Raff, Maya Fuchs, Townsend Southard Pantano, James Holt, and Kristopher Micinski. Assemblage: Automatic Binary Dataset Construction for Machine Learning. In A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang, editors, *Advances in Neural Information Processing Systems*, volume 37, pages 58698–58715. Curran Associates, Inc., 2024. URL https://proceedings.neurips.cc/paper_files/paper/2024/file/6bbefc73a187dd42e0dc065b4e7a0615-Paper-Datasets_and_Benchmarks_Track.pdf.

Louis Milliken, Sungmin Kang, and Shin Yoo. Beyond pip install: Evaluating LLM Agents for the Automated Installation of Python Projects, December 2024. URL http://arxiv.org/abs/2412.06294. arXiv:2412.06294 [cs].

Kevin Agina Onyango and Geoffrey Wambugu Mariga. Comparative Analysis on the Evaluation of the Complexity of C, C++, Java, PHP and Python Programming Languages based on Halstead Software Science. *International Journal of Computer and Information Technology(2279-0764)*, 12 (1), March 2023. ISSN 2279-0764. doi: 10.24203/ijcit.v12i1.294. URL https://www.ijcit.com/index.php/ijcit/article/view/294. Number: 1.

Kuntal Kumar Pal, Ati Priya Bajaj, Pratyay Banerjee, Audrey Dutcher, Mutsumi Nakamura, Zion Leonahenahe Basque, Himanshu Gupta, Saurabh Arjun Sawant, Ujjwala Anantheswaran, Yan Shoshitaishvili, Adam Doupé, Chitta Baral, and Ruoyu Wang. "Len or index or count, anything but v1": Predicting Variable Names in Decompilation Output with Transfer Learning. In *2024 IEEE Symposium on Security and Privacy (SP)*, pages 4069–4087, May 2024. doi: 10.1109/SP54263.2024.00152. URL https://ieeexplore.ieee.org/document/10646727. ISSN: 2375-1207.

Juan Altmayer Pizzorno and Emery D. Berger. CoverUp: Effective High Coverage Test Generation for Python, May 2025. URL http://arxiv.org/abs/2403.16218. arXiv:2403.16218 [cs].

Noah Shinn, Federico Cassano, Edward Berman, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language Agents with Verbal Reinforcement Learning, October 2023. URL http://arxiv.org/abs/2303.11366. arXiv:2303.11366 [cs].

Yifan Song, Guoyin Wang, Sujian Li, and Bill Yuchen Lin. The Good, The Bad, and The Greedy: Evaluation of LLMs Should Not Ignore Non-Determinism, July 2024. URL http://arxiv.org/abs/2407.10457. arXiv:2407.10457 [cs].

Hongjin Su, Ruoxi Sun, Jinsung Yoon, Pengcheng Yin, Tao Yu, and Sercan Ö Arık. Learn-by-interact: A Data-Centric Framework for Self-Adaptive Agents in Realistic Environments, January 2025. URL http://arxiv.org/abs/2501.10893. arXiv:2501.10893 [cs].

Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. LLaMA: Open and Efficient Foundation Language Models, February 2023. URL http://arxiv.org/abs/2302.13971. arXiv:2302.13971 [cs].

Zora Zhiruo Wang, Jiayuan Mao, Daniel Fried, and Graham Neubig. Agent Workflow Memory, September 2024. URL http://arxiv.org/abs/2409.07429. arXiv:2409.07429 [cs].

Danning Xie, Zhuo Zhang, Nan Jiang, Xiangzhe Xu, Lin Tan, and Xiangyu Zhang. ReSym: Harnessing LLMs to Recover Variable and Data Structure Symbols from Stripped Binaries. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, CCS '24, pages 4554–4568, New York, NY, USA, December 2024. Association for Computing Machinery. ISBN 9798400706363. doi: 10.1145/3658644.3670340. URL https://dl.acm.org/doi/10.1145/3658644.3670340.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. ReAct: Synergizing Reasoning and Acting in Language Models, March 2023. URL http://arxiv.org/abs/2210.03629. arXiv:2210.03629 [cs].

Shunyu Yao, Noah Shinn, Pedram Razavi, and Karthik Narasimhan. $$-bench: A Benchmark for Tool-Agent-User Interaction in Real-World Domains, June 2024. URL http://arxiv.org/abs/2406.12045. arXiv:2406.12045 [cs].

Zhiqiang Yuan, Yiling Lou, Mingwei Liu, Shiji Ding, Kaixin Wang, Yixuan Chen, and Xin Peng. No More Manual Tests? Evaluating and Improving ChatGPT for Unit Test Generation, May 2024. URL http://arxiv.org/abs/2305.04207. arXiv:2305.04207 [cs].

A Filtering Keywords

A portion of keywords we used to filter out low-quality OSS including:

homework, assignment, tutorial, exercise, solution, course, student, university, college, class, lecture, demo, practice, presentation, getting started, hello world, starter code, sample code, example code, documentation.

B Sample Size Estimation

To estimate the minimum sample size required to measure a population proportion with 95% confidence and a margin of error of 5%, the standard formula for proportion estimation is:

$$n_0 = \frac{Z^2 p(1-p)}{E^2}, \tag{1}$$

where $Z = 1.96$ for 95% confidence interval, $E = 0.05$ is the error margin, and $p = 0.5$ is chosen to maximize variance (i.e., yield the largest conservative sample size). This gives:

$$n_0 = \frac{1.96^2 \times 0.5 \times 0.5}{0.05^2} = 384.16.$$

For finite populations, we apply the finite population correction (FPC)Cochran [1977]:

$$n = \frac{n_0}{1 + \frac{n_0 - 1}{N}} = \frac{384.16}{1 + \frac{383.16}{6{,}568{,}809}} \approx 384.14. \tag{2}$$

$$n = \frac{n_0}{1 + \frac{n_0 - 1}{N}} = \frac{384.16}{1 + \frac{383.16}{57{,}572}} \approx 384.14. \tag{3}$$

We round up to obtain a final required sample size of $n = 385$. We accordingly conduct a random sample of 385 repositories from the previously mentioned corpus to compose the final test set.

C LLM prompts

C.1 LLM baseline prompts

Patch proposed by Agent

```
You are an expert Linux build engineer working inside a
 ↪  **UbuntuâĂŚbased Docker container**. The pre-installed
 ↪  software and libraries are as listed in the following
 ↪  dockerfile content:
\{per\_installed\_libraries\_in\_docker\}

\#\#\# Your task
Generate a **sequence of Bash commands** (one command per
 ↪  line, no comments, no explanations) that will:
1. Install every buildâĂŚtime dependency needed to compile
 ↪  the repository **{repo\_full\_name}** that lives at
 ↪  **{repos\_dir\_in\_docker}**.
âĂć Use nonâĂŚinteractive `apt-get update \&\& apt-get
 ↪  install -y âĂę` when possible.
âĂć Avoid PPAs unless strictly necessary.
âĂć Assume you run as root, so no `sudo` is required.
2. **Detect build system and configure debug build:**
Examine the repository structure (files listed below) to
 ↪  choose the proper build configuration command. Configure
 ↪  the build system in Debug mode (i.e., include DWARF
 ↪  symbols, disable optimizations).
3. **Install the main binary:**
Identify the primary or main binary (for example, the one
 ↪  built from the projectâĂŹs main executable) and install
 ↪  it into {repos\_dir\_in\_docker}
- Ensure the installation directory exists (create it if
 ↪  necessary with `mkdir -p`).
- Copy the main binary into that directory and set executable
 ↪  permissions if needed.

\#\#\# Strict requirements:
***Output only Bash commands, separated using the newline
 ↪  character.**
Do not provide any explanations, markdown, or extra comments.
* The commands must be **fully sequential and ready-to-run**
 ↪  when concatenated.
There should be no interactive prompts or assumptions beyond
 ↪  what is provided.
* All steps must run successfully in a typical Docker Ubuntu
 ↪  environment.
* Assume the current working directory is ** "/app" **.

\#\#\# Repository context
**Repo name:** {repo\_full\_name}
**Root path in container:** {repos\_dir\_in\_docker}

**README:**
{readme\_content}

**TopâĂŚlevel file list:**
{files\_in\_root\_dir}
```

C.2 System Prompt for *Bash Command Generator*

Patch proposed by Agent

```
You are an helpful AI assistant that is an expert in
 ↪  compiling cloned GitHub repositories and handling
 ↪  compilation errors during the process by generating bash
 ↪  commands.
The current working directory is `/app`, and all commands
 ↪  must use absolute paths referencing the repository's
 ↪  specific clone directory, with no placeholders. The
 ↪  compilation process runs inside a Docker container with
 ↪  root access, so do not use `sudo`. Your suggested code
 ↪  must be complete and executable, as the user cannot
 ↪  modify it. Ensure the target repository is compiled with
 ↪  debug information, for instance by adding `-g -O0` to
 ↪  compiler flags, and do not strip this information after
 ↪  compilation. Whenever possible, use a prefix or `DESTDIR`
 ↪  flag during the `make` command to save compiled artifacts
 ↪  inside the clone directory. Always run `make install`
 ↪  after compilation, using multiple cores to speed up the
 ↪  process, but do not run `make check` or `make test`. More
 ↪  detailed building instructions from the repository will
 ↪  be provided, which you must follow. You should attempt to
 ↪  fix any errors that occur. To end the process upon
 ↪  success or failure, send a message explaining the reason
 ↪  followed by the word "terminate," but never include
 ↪  "terminate" in a response that also contains a code
 ↪  block. Do not show appreciation in your responses; if
 ↪  "Thank you" is said, reply only with "TERMINATE".
```

C.3 System Prompt for *Executor Agent*

System Prompt:
You are an AI assistant that can run bash commands or execute function calling and conduct the process of GitHub repository compilation.

D Case Study 1: Agentic Compilation patching source files

During our log analysis we observe that in some cases Agentic Compilation would attempt to fix the source files after encountring compilation errors and then continue building the project. *s9xie/hed* repository, part of BUILD-BENCHhas a code base that is 10 years old which relies on outdated packages and dependencies. It uses OpenCV v3 API calls and originally build to run on Ubuntu 14. Newer versions of OpenCV v4 updated their API, which causes this project to fail to build out-of-the-box on recent versions of Ubuntu. Based on error log that Agent received as part of feedback loop, it automatically patched the source files updating the occurrences of old API and successfully compiled the repository. For instance, it updated `CV_LOAD_IMAGE_COLOR` to `IMREAD_COLOR`. It showcases the potential of AI-based compilation method of patching deemed 'uncompilable' repositories, whereas a rule-based approach would never be able to fix it automatically without human-assistance.

Error Log

```
/app/k8s_compiled_repos/hed/src/caffe/layers/window_data_lay⌋
↪  er.cpp: In member function âĂŸvirtual void
↪  caffe::WindowDataLayer<Dtype>::load_batch(caffe::Batch<D⌋
↪  type>*)âĂŹ:
/app/k8s_compiled_repos/hed/src/caffe/layers/window_data_lay⌋
↪  er.cpp:288:42: error: âĂŸCV_LOAD_IMAGE_COLORâĂŹ was not
↪  declared in this scope
  288 |         cv_img = cv::imread(image.first,
   ↪  CV_LOAD_IMAGE_COLOR);
      |
       ↪  ^~~~~~~~~~~~~~~~~~
...
...
...
/app/k8s_compiled_repos/hed/src/caffe/util/io.cpp: In
↪  function âĂŸcv::Mat caffe::ReadImageToCVMat(const
↪  string&, int, int, bool)âĂŹ:
/app/k8s_compiled_repos/hed/src/caffe/util/io.cpp:73:34:
↪  error: âĂŸCV_LOAD_IMAGE_COLORâĂŹ was not declared in this
↪  scope
   73 |   int cv_read_flag = (is_color ? CV_LOAD_IMAGE_COLOR
    ↪  :
      |                                  ^~~~~~~~~~~~~~~~~~
/app/k8s_compiled_repos/hed/src/caffe/util/io.cpp:74:5:
↪  error: âĂŸCV_LOAD_IMAGE_GRAYSCALEâĂŹ was not declared in
↪  this scope
   74 |     CV_LOAD_IMAGE_GRAYSCALE);
      |     ^~~~~~~~~~~~~~~~~~~~~~
...
...
/app/k8s_compiled_repos/hed/src/caffe/layers/window_data_lay⌋
↪  er.cpp: In member function âĂŸvirtual void
↪  caffe::WindowDataLayer<Dtype>::load_batch(caffe::Batch<D⌋
↪  type>*)âĂŹ:
```

Patch proposed by Agent

Agent Compilation :

```bash
# Patch the OpenCV constants to use new names              \
↪  (IMREAD_COLOR and IMREAD_GRAYSCALE) in the affected    \
↪  source files.
sed -i 's/CV_LOAD_IMAGE_COLOR/IMREAD_COLOR/g'              \
↪  /app/k8s_compiled_repos/hed/src/caffe/util/io.cpp
sed -i 's/CV_LOAD_IMAGE_GRAYSCALE/IMREAD_GRAYSCALE/g'      \
↪  /app/k8s_compiled_repos/hed/src/caffe/util/io.cpp
sed -i 's/CV_LOAD_IMAGE_COLOR/IMREAD_COLOR/g'              \
↪  /app/k8s_compiled_repos/hed/src/caffe/layers/window_⌋  \
↪  data_layer.cpp
```

E CASE STUDY 2: CMake Error due to Directory Changes

Agents fail to compile repositories due to various reasons, as described in 6.3. For instance, when compiling GitHub repository *bernhard-schmitzer/optimal-transport*, the agent invoked `cmake` from the wrong working directory; further troubleshooting would have revealed that the `CMakeLists.txt` resides in the `src` subdirectory. While changing to a different directory is a simple and straghtforward solution, but the agent did not perform sufficient troubleshooting.

Compilation commands

```
```bash
cd /app/k8s_compiled_repos/optimal-transport/v0.2.0
mkdir -p build && cd build
cmake -DCMAKE_BUILD_TYPE=Debug -DCMAKE_CXX_FLAGS="-g -O0"  \
 ↪  -DCMAKE_C_FLAGS="-g -O0" ..
make -j15
make install                                               \
 ↪  DESTDIR=/app/k8s_compiled_repos/optimal-transport

```
```
```

Execution log

```
exitcode: 2 (execution failed)
CMake Error: The source directory                          \
 ↪  "/app/k8s_compiled_repos/optimal-transport/v0.2.0"     \
 ↪  does not appear to contain CMakeLists.txt.
```

F CASE STUDY 3: Insufficient Troubleshooting

Agents fail to compile repositories due to various reasons, as described in 6.3. For instance, when compiling GitHub repository *blitz3d-ng/package* , the agent attempts to first initialize the Git submodules and the execute make to compile the repository. However, the execution failed. First, the current user running the git submodule update command is not the owner of the specified repository directory, causing *dubious ownership* error. Then this results in a follow-up error in make that indicates *No rule to make target 'install'*.

To resolve the error, the agent must address the **submodule initialization error**, as it is the **root cause**. If the agent focuses only on tackling the follow-up `make` error, the resolution will ultimately fail, highlighting the need for agents to diagnose and fix the primary issue in the execution chain.
```

Compilation commands

```bash
apt-get update
apt-get install -y git ninja-build libxml2-dev zlib1g-dev  \
 ↪  build-essential autoconf libtool gettext autopoint     \
 ↪  gperf cmake clang libwxgtk3.0-gtk3-dev libxrandr-dev   \
 ↪  libxinerama-dev libxcursor-dev uuid-dev                \
 ↪  libfontconfig1-dev
cd /app/compiled_repos/package
git submodule update --init --recursive
source /app/k8s_compiled_repos/package/env.sh
make -j15 llvm
make -j15 CPPFLAGS='-g -O0' CXXFLAGS='-g -O0'
make install PREFIX=/app/compiled_repos/package
```

Execution log

```
exitcode: 2 (execution failed)
git submodule update failed:
fatal: detected dubious ownership in repository at         \
 ↪  '/app/k8s_compiled_repos/package'

...


make install failed: No rule to make target 'install'
```