CompileAgent: Automated Real-World Repo-Level Compilation with Tool-Integrated LLM-based Agent System

Anonymous ACL submission

Abstract

With open-source projects growing in size and complexity, manual compilation becomes tedious and error-prone, highlighting the need for automation to improve efficiency and accuracy. However, the complexity of compilation instruction search and error resolution makes automatic compilation challenging. Inspired by the success of LLM-based agents in various fields, we propose CompileAgent, the first LLM-based agent framework dedicated to repo-level compilation. CompileAgent integrates five tools and a flow-based agent strategy, enabling interaction with software artifacts for compilation instruction search and error resolution. To measure the effectiveness of our method, we design a public repo-level benchmark CompileAgentBench, and we also design two baselines for comparison by combining two compilation-friendly schemes. The performance on this benchmark shows that our method significantly improves the compilation success rate, ranging from 10% to 71%. Meanwhile, we evaluate the performance of CompileAgent under different agent strategies and verify the effectiveness of the flow-based strategy. Additionally, we emphasize the scalability of CompileAgent, further expanding its application prospects.

1 Introduction

Compilation is the process of converting source code into executable files or libraries. Currently, many open-source tool libraries and application software projects can be used directly after compiling into executable files or libraries. Not only that, these files or libraries can also be used for subsequent work, including building diverse datasets (Ye et al., 2023), conducting performance testing and optimization (Tan et al., 2020), security and vulnerability analysis (Jiang et al., 2024), etc.

For single-file compilation, the compiler only needs to process a single source code file and generate the corresponding target code. However, compiling an open-source code repository shared by others is a far more complex, time-consuming (Wang et al., 2024b) and demanding task in actual software engineering. This process goes beyond handling the source code itself and requires addressing intricate challenges such as environment adaptation, dependency management, and build configuration. As a result, developers tend to spend most of their time troubleshooting challenges during the compilation process.

To date, no research has specifically focused on how to achieve automated compilation at the repository level. Drawing from developers' experience in compiling code repositories, we identify two core challenges in this task. The first is the discovery and accurate extraction of compilation instructions from repositories, which often involve varied build systems, scripts, and configurations. The second challenge is resolving compilation errors encountered during the process, which is required to address issues such as dependency conflicts, environment mismatches, and code compatibility.

Recently, the application of LLM-based agents for automating complex tasks has gained significant attention across various fields. They have been successfully employed in areas such as code generation (Huang et al., 2023; Zhang et al., 2024a), bug fixing (Liu et al., 2024b; Bouzenia et al., 2024), and penetration testing (Deng et al., 2024; Shen et al., 2024; Bianou and Batogna, 2024), where they autonomously perform tasks that traditionally require human intervention. Inspired by the success of these applications, we propose leveraging agents for the automation of repository-level compilation tasks. By doing so, we aim to streamline the compilation process, reduce manual intervention, and address the challenges inherent in compiling open-source repositories.

In this paper, we propose CompileAgent, the first novel approach that leverages LLM-based agents for automated repo-level compilation. To address the two key challenges identified earlier, we have designed five specialized tools and a flow-based agent strategy. CompileAgent can effectively complete the compilation of code repositories by interacting with external tools. To evaluate the effectiveness of our approach, we manually constructed CompileAgentBench, a benchmark designed for repository compilation. This benchmark consists of 100 repositories in C and C++, sourced from Github. We further conducted comprehensive experiments to evaluate the performance of CompileAgent by applying it to seven well-known LLMs, with parameter sizes ranging from 32B to 236B, to demonstrate its broad applicability. When compared to the existing baselines, CompileAgent achieved a notable increase in compilation success rates across all LLMs, with improvements reaching up to 71%. Additionally, the total compilation time can be reduced by up to 121.9 hours, while maintaining a low cost of only \$0.22 per project. We compared the flow-based strategy with several other strategies suitable for the compilation task, further validating its effectiveness. Moreover, we conducted ablation experiments to validate the necessity of each component within the system. These experiments provide strong evidence that CompileAgent effectively addresses the challenges of code repository compilation.

Our contributions can be summarized as follows:

- We make the first attempt to explore repo-level compilation by LLM-based agent, offering valuable insights into the practical application of agents in real-world scenarios.
- We propose CompileAgent, a LLM-based agent framework tailored for the repo-level compilation task. By incorporating five specialized tools and a flow-based agent strategy, the framework enables LLMs to autonomously and effectively complete the compilation of repositories.
- We construct CompileAgentBench, a benchmark for compiling code repositories that includes high-quality repositories with compilation instructions of varying difficulty and covering a wide range of topics.
- Experimental results on seven LLMs demonstrate the effectiveness of CompileAgent in compiling code repositories, highlighting the potential of agent-based approaches for tackling complex software engineering challenges.

2 Background

2.1 LLMs and Agents

LLMs have demonstrated remarkable performance across a wide range of Natural Language Processing (NLP) tasks, such as text generation, summarization, translation, and question-answering. Their ability to understand and generate human-like text makes them a powerful tool for various applications. However, LLMs are limited to NLP tasks and struggle with tasks that involve direct interaction with the external environment.

Recent advancements in LLMs have significantly expanded their capabilities, with many models now supporting function calls as part of their core functionalities. This enhancement allows LLMs to dynamically interact with external systems and tools, playing a key role in the development of the AI agents (Qian et al., 2024b; Islam et al., 2024; Huang et al., 2024; Qian et al., 2024a; Chen et al., 2023; Xie et al., 2023). Nowadays, with the popularity of agent-based frameworks, researchers have begun to develop agent-based methods to solve complex tasks, such as OpenHands (Wang et al., 2024e), AutoCodeRover (Zhang et al., 2024b), and SWE-Agent (Yang et al., 2024).

2.2 Automatic Compilation

In modern software development, there are a large number of open-source code repositories, but due to differences in project management and document writing among developers, the quality and standardization of compilation guides vary. Many projects lack detailed compilation instructions, which may cause users to encounter problems such as inconsistent environment configuration or lack of necessary dependencies when trying to compile. In addition, some open-source projects store compilation guides in external documents or websites without clearly marking them in the codebase, resulting in the compilation process that relies on manual steps, which is both error-prone and time-consuming. These problems make it more challenging to automate the compilation of open-source projects, and also highlight the importance of automated compilation tools in improving the maintainability and scalability of open-source projects.

Oss-Fuzz-Gen(Liu et al., 2024a) is an open-source tool designed to fuzz real-world projects, including a part for building projects. This part works by analyzing the structure of the code repository and searching for specific files. Based

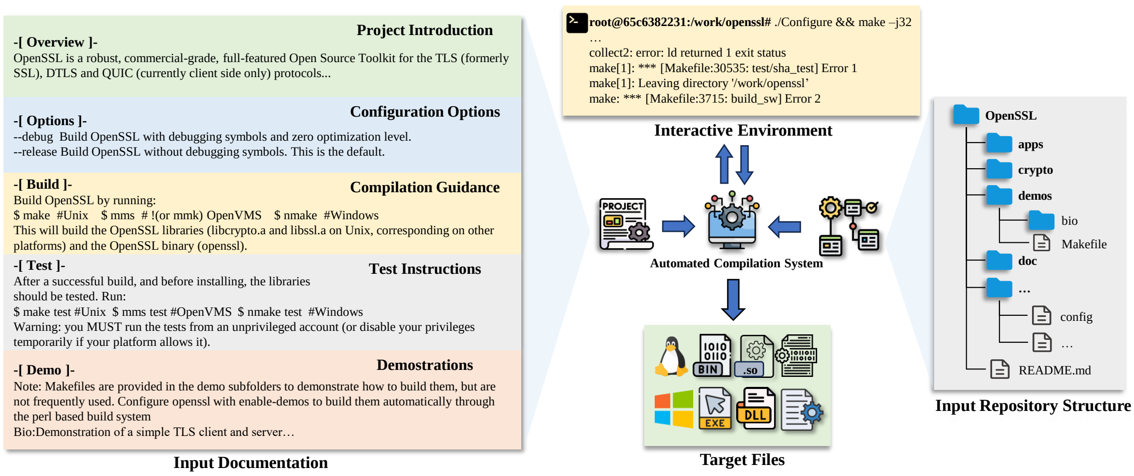

Figure 1: An illustrative example of the automated repo-level compilation. The task input contains code repository documentation and structure, and the automated compilation system can interact with the interactive environment.

on the presence of these files, a set of predefined compilation instructions is then executed to build the project. For example, if the repository contains `bootstrap.sh` and `Makefile.am`, Oss-Fuzz-Gen will execute the "`./bootstrap.sh; ./configure; make`" commands in sequence to build the project. However, Oss-Fuzz-Gen may not be sufficient for projects where the specified files are absent. Additionally, the tool lacks adaptability to changing environments, making it less flexible in dynamic or evolving software projects.

To be closer to realistic compilation scenarios, we formalize repo-level compilation tasks and propose CompileAgent to help LLMs complete this complex task. We also built a repo-level compilation benchmark CompileAgentBench to evaluate our approach and provide details of the benchmark in Appendix A. Compared with Oss-Fuzz-Gen, CompileAgent is more suitable for handling real-world compilation tasks.

3 Repo-Level Compilation Task

To bridge the gap between current compilation tasks and real-world software building scenarios, we formalized the repo-level compilation task. Unlike simple file-level compilation, code repositories often entail complex build configurations and interdependencies across multiple files. Consequently, an automated compile system as shown in Figure 1, which is an integrated tool or a comprehensive framework designed to facilitate the entire compilation process, must comprehend the entire repository, its dependencies, and the interactions between its components to ensure successful compilation at the repo-level. The repo-level compilation task focuses on managing the compilation process by considering all relevant software artifacts within the repository, including documentation, repository structure, and interactive environment.

Documentation. It provides essential insights into the project, including project introduction, configuration options, compilation guidelines, testing frameworks, and Demonstrations. Automated compile system can leverage it to extract and interpret information necessary for accurately configuring and executing the compilation process. Moreover, documentation often contains nuanced details about platform-specific dependencies or build settings that are critical for success.

Repository Structure. The structure of a repository reflects the organization and relationships among its files and modules. Effective repo-level compilation depends on a deep understanding these relationships, including dependency hierarchies between files or modules, and adhering to build sequence constraints(e.g., resolving "`cmake`" configurations before invoking "`make`"). Furthermore, addressing external library dependencies, such as linking with libraries like OpenSSL or Boost, is crucial for ensuring both compatibility and correctness. Efficiently navigating this structure is pivotal for repositories with intricate interdependencies.

Interactive Environment. The interactive environment is integral to successful repo-level compilation, as it provides essential support throughout the process. It can provide detailed error messages and diagnostic information to the automated compile system during the compilation process, allowing it to identify and resolve issues in real time. This dynamic feedback loop allows the automated compile system to adjust the compilation process as needed, ensuring greater accuracy and efficiency.

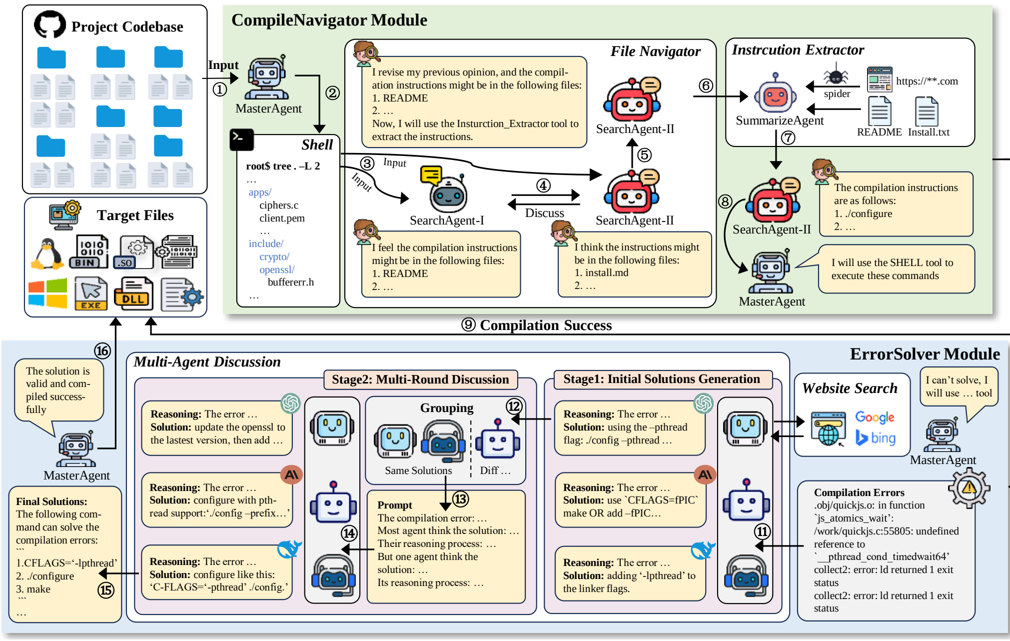

Figure 2: The overview of CompileAgent. By providing the repository of a given project, the automated compilation process can be seamlessly completed using the designed modules and agent strategy. Agents not explicitly specified are driven by DeepSeek-v2.5.

Additionally, the interactive environment should isolate the compilation process to safeguard the physical machine and provide independent build environments for each project.

In this paper, we consider LLM-based agent as an automated compilation system. Our objective is to rigorously evaluate its effectiveness in automating the repo-level compilation, ensuring that it can accurately identify the correct compilation instructions and efficiently resolve any issues that arise during the compilation process.

4 Method

In this section, we present the design of the LLM-based agent framework, CompileAgent, aimed at automating repo-level compilation. To effectively address the two key challenges mentioned in Section 1, we design two core modules, CompileNavigator and ErrorSolver, which together include five supporting tools, all integrated into a flow-based agent strategy, as shown in Figure 2.

4.1 Designed Module

When searching for compilation instructions in the given code repository, users typically rely on the repository's structure to identify potential files containing the necessary instructions. Moreover, when encountering difficulties during the compilation process that are hard to resolve, they often seek solutions through online resources, LLMs or other methods. To locate compilation instructions and resolve compilation errors, we model the process of solving the challenges and design the following two modules.

4.1.1 CompileNavigator

The CompileNavigator module is designed to tackle the challenge of finding the correct compilation instructions within a code repository. Typically, the necessary instructions are scattered across different documentation types, such as `README`, `doc.html`, `install.txt`, etc. making it difficult to locate them quickly. To address this challenge, the module employs three key tools: Shell, File Navigator, and Instruction Extractor.

Shell. To ensure the security of physical machine during the compilation process, we isolate the entire compilation workflow from the host system by creating a container using Docker. The downloaded project is mounted into this container, and an SSH connection is established to access the terminal shell. The Docker container is built on the Ubuntu 22.04 operating system image. Through this tool, LLMs can interact with the interactive environment and execute any necessary commands.

File Navigator. To accurately locate the file containing the compilation instructions, we design two agents, SearchAgent I and SearchAgent II. The repository's structural information is provided as input, and the two agents engage in a collaborative discussion to determine the most likely file containing the compilation instructions.

Instruction Extractor. After identifying the files that likely contain the compilation instructions, the next task is to extract the instructions from them. In order to complete this, we design the SummarizeAgent, which reads the content of a specified file and searches for URLs related to compilation instructions within the file. If such URLs are found, requests are sent to retrieve the web page content. Finally, SummarizeAgent summarizes and outputs the relevant compilation instructions.

4.1.2 ErrorSolver

The ErrorSolver module is designed to address compilation errors during the project build process, which can stem from various issues such as syntax problems, missing dependencies, or configuration conflicts. To resolve these errors, we develop two key tools in this module: Website Search and Multi-Agent Discussion.

Website Search. Developers frequently publish solutions to compilation problems on websites, which search engines treat as valuable knowledge databases. When faced with similar problems, users can submit queries to search engines to find relevant solutions. Inspired by this, we encapsulate Google Search[1] engine into a tool. However, since search results may include irrelevant content, we instruct the agents using the tool to prioritize reliable, open-source websites, like Github[2] and StackOverflow[3], and then aggregate the relevant information to provide a solution to the user's query.

Multi-Agent Discussion. Although there are various single-agent approaches exist for solving reasoning tasks, such as self-polishing (Xi et al., 2023b), self-reflection (Yan et al., 2024), self-consistency (Wang et al., 2024a) and selection-inference (Creswell et al., 2022), we think these complex reasoning approaches are unnecessary for solving compilation errors. Compilation errors typically come with clear error messages, such as path or environment configuration issues and compatibility problems. These errors can generally be resolved through straightforward analysis, consulting documentation, and making reasonable inferences, without the need of advanced reasoning processes. Inspired by Wang et al. (Wang et al., 2024d) and reconcile (Chen et al., 2024), we propose a Multi-Agent Discussion approach specifically designed to address compilation errors. In this method, multi-agents first analyze the complex compilation error and generate an initial solution. The agents then enter a multi-round discussion phase, where each can revise its analysis and response based on the inputs from the other agents in the previous round. The discussion continues until a consensus is reached or for up to R rounds. At the end of each round, the solutions, consisting of command lines, are segmented, and repeated terms are counted. If the number of repeated terms exceeds a specified threshold, the solutions are considered equivalent, and a final team response is generated. In this paper, we set up three agents for the discussion, with a maximum of 3 Rounds.

4.2 Agent Strategy

When compiling a given project, users typically begin by consulting the project's compilation guide, and then execute the relevant compilation commands based on their environment. If issues arise during the process, they often resort to online searches or query tools like ChatGPT to troubleshoot until the compilation succeeds. Inspired by this workflow, to enable LLMs to effectively leverage our designed tools, we propose a flow-based agent strategy tailored for the automated compilation task.

The strategy defines the sequence in which tools are used and connects them seamlessly through prompts. MasterAgent is responsible for invoking the tools. The process is as follows:

① MasterAgent begins by downloading the target code repository to the local system and mounting it into the container using the Shell tool;

② Next, MasterAgent uses the Shell tool to run commands like "`tree`" within the container to retrieve the repository structure;

③ Then, MasterAgent invokes the FileNavigator tool to identify files that may contain the necessary compilation instructions;

④ Subsequently, MasterAgent uses the InstructionExtractor tool to extract the compilation instructions and execute them via the Shell tool;

⑤ If the Shell tool returns a successful compilation result, the compilation process is complete. If a compilation error occurs, MasterAgent first attempts to resolve the issue independently. If the issue persists after attempts, the ErrorSolver module is activated for several rounds of collaborative discussion. Finally, the compilation status is determined based on the Shell tool's outcome.

[1] https://www.google.com/
[2] https://github.com/
[3] https://stackoverflow.com/

5 Experiment

We conduct extensive experiments to answer three research questions: (1) How much does CompileAgent improve the project compilation success rate compared to existing methods? (2) How effective is the flow-based strategy we designed when compared to existing agent strategies? (3) To what extent do the tools integrated within CompileAgent contribute to successful repo-level compilation?

5.1 Experimental Setup

Benchmark. To the best of our knowledge, there is no existing work that specifically evaluates repo-level compilation. Therefore, we manually construct a new benchmark for repo-level compilation to evaluate the effectiveness of our approach in this domain. We select 100 projects from many C/C++ projects on Github and carefully consider multiple factors during the project selection to ensure the authority and diversity of CompileAgentBench. First, we screen the projects based on the number of stars to ensure that the selected projects have high representativeness and practical value in the community. Moreover, we also consider the topics involved in the projects and finally select projects covering 14 different fields, including areas such as crypto, audio, and neural networks. On this basis, we also pay special attention to whether each project provided a clear compilation guide. Meanwhile, we arrange for three participants with 3 to 4 years of project development experience to manually compile these 100 projects to further verify the compilability of the selected projects and the accuracy of the evaluation. We finally obtain the target files of these 100 projects, and the entire compilation process took about 46 man-hours. More details refer to Appendix A.

Baselines. As the first work dedicated to automating repo-level compilation, there is no related work for us to compare except Oss-Fuzz-Gen. However, there are some projects or technologies that are helpful for automated compilation tasks, such as the Readme-AI[4] project and Retrival-Augumented Generation (RAG) techniques.

Readme-AI is a developer tool that can generate well-structured and detailed documentation for a code repository based solely on its URL or file path. For cost-effectiveness, we utilize GPT-4o mini for documentation generation and specify in the requirements that the "`How to compile/build from source code`" section should be included. A detailed example of this process is provided in Appendix B. RAG refers to a technique that enhances the output of LLMs by allowing them to reference external knowledge sources during response generation. In the compilation task, we leverage RAG as a tool. Specifically, we traverse the possible compilation files in the code repository, and then cut these file contents into chunks and generate vector embeddings. Each time the compilation instructions are searched for, LLMs generate instructions by retrieving the vector database. For a specific example, please refer to Appendix C.

We also compare the flow-based agent strategy designed in this paper with existing agent strategies. According to the research of Wang et al. (Wang et al., 2024c) and Xi et al. (Xi et al., 2023a), we select two common agent strategies that are suitable for the automated compilation task, including ReAct (Yao et al., 2022), Plan-and-Execute (Wang et al., 2023). In addition, we also consider the comparison with OpenAIFunc (OpenAI, 2023).

Base LLMs. We apply CompileAgent to seven advanced LLMs, including three closed-source LLMs, i.e., GPT-4o (GPT-4o, 2024), Claude-3-5-sonnet (Claude, 2024), Gemini-1.5-flash (Gemini, 2024), as well as four open-source LLMs, i.e., Qwen2.5-32B-Instruct (Team, 2024), Mixtral-8×7B-Instruct (MistralAI, 2023), LLama3.1-70B-Instruct (Meta-LLaMa, 2024), DeepSeek-v2.5 (DeepSeek-AI, 2024). Additional descriptions are provided as a part of Table 1.

Metrics. In order to comprehensively evaluate the effectiveness of automated compilation tasks, we select three key indicators: compilation success rate, time cost, and expenses. Among these, the compilation success is determined when the target files in the precompiled projects completely match those generated by CompileAgent.

5.2 Repo-Level Compilation Performance

In this experiment, we use the specially designed repo-level benchmark, CompileAgentBench, to evaluate the performance of CompileAgent and three baselines in compiling code repositories across seven well-known LLMs. The results are presented in Table 1.

It turns out that our proposed CompileAgent-Bench is more challenging when not using LLMs methods, as evidenced by the lower compilation success rate of Oss-Fuzz-Gen. Compared with existing baselines, CompileAgent has significant

[4] https://github.com/eli64s/readme-ai

Table 1: The results of different baselines on CompileAgentBench.

Models	Size	Oss-Fuzz-Gen[1]			Readme-AI			RAG			CompileAgent		
		Csr[2]	$Time$[3]	Exp[4]	Csr	$Time$	Exp	Csr	$Time$	Exp	Csr	$Time$	Exp
Closed-source LLMs													
GPT-4o (GPT-4o, 2024)	-				72%	128.80	42.94	67%	11.12	45.78	89%	8.38	16.53
Claude-3-5-sonnet (Claude, 2024)	-	25%	53.01	-	79%	127.33	55.26	78%	8.30	54.44	96%	5.37	22.02
Gemini-1.5-flash (Gemini, 2024)	-				41%	123.68	32.37	46%	9.28	35.72	65%	3.55	2.39
Open-source LLMs													
Qwen2.5-32B-Instruct (Team, 2024)	32B				70%	127.82	33.18	62%	10.55	36.73	80%	5.25	3.16
Mixtral-8×7B-Instruct (MistralAI, 2023)	42B	25%	53.01	-	38%	124.60	33.12	45%	10.82	36.49	55%	4.88	4.32
LLama3.1-70B (Meta-LLaMa, 2024)	70B				61%	125.03	33.57	61%	10.98	36.87	79%	7.38	2.71
DeepSeek-v2.5 (DeepSeek-AI, 2024)	236B				71%	125.43	33.70	72%	11.30	36.08	91%	11.38	3.31

[1] The Oss-Fuzz-Gen project operates without relying on LLMs.
[2] The proportion of successfully compiled projects to all projects.
[3] The total duration required to complete the compilation process, measured in hours.
[4] The total expense incurred during the compilation process, measured in US dollars.

Table 2: The results of different agent strategies on CompileAgentBench.

Models	Size	OpenAIFunc[1]			PlanAndExecute			ReAct			Flow-based		
		Csr	$Time$	Exp	Csr	$Time$	Exp	Csr	$Time$	Exp	Csr	$Time$	Exp
Closed-source LLMs													
GPT-4o (GPT-4o, 2024)	-	80%	6.75	22.51	40%	5.18	10.02	72%	6.58	23.63	89%	8.38	16.53
Claude-3-5-sonnet (Claude, 2024)	-	-	-	-	72%	5.02	13.77	81%	8.40	25.26	96%	5.37	22.02
Open-source LLMs													
LLama3.1-70B (Meta-LLaMa, 2024)	70B	-	-	-	26%	4.77	2.14	49%	10.48	6.52	79%	7.38	2.71
DeepSeek-v2.5 (DeepSeek-AI, 2024)	236B	-	-	-	70%	6.72	1.42	78%	11.32	3.88	91%	11.38	3.31

[1] The openaifunc refers to OpenAI's LLMs equipped with the capability to invoke functions.

performance improvements on LLMs with various sizes. Specifically, CompileAgent achieves the highest performance on the Claude-3-5-sonnet model, improving by 71%, 17%, and 18% over all baselines, respectively; in terms of time cost, it saves 47.64 hours, 121.96 hours, and 2.93 hours; in terms of expenses, the average cost per project is only $0.22. Excluding Oss-Fuzz-Gen, the total cost is reduced by $33.24 and $32.42, respectively. The performance improvement on other LLMs ranges from 30% to 71%, 10% to 24%, and 10% to 22%, which clearly demonstrates the effectiveness of our method. This indicates that the integrated tools in CompileAgent can effectively assist LLMs in completing the compilation process, meeting the real-world needs of repo-level compilation.

In addition, we also find that the more advanced LLMs tend to show better performance with CompileAgent. However, for the poor performance of Mixtral-8×7B-Instruct, we speculate that may be related to its model architecture design.

5.3 Strategy Performance

We also evaluate the impact of different agent strategies on CompileAgent, and make slight modifications to other strategies, enabling them to call the tool we designed. Additionally, we strategically select a set of representative LLMs for evaluation, considering the constraints of available resources and computing power. Table 2 summarizes the experimental results of the evaluation.

Our flow-based agent strategy achieves the highest compilation success rate on Claude-3-5-sonnet, but it also brings a lot of costs. It is worth noting that the success rate of each compilation strategy generally decreases when using LLMs with fewer parameters. Despite this, our designed strategy can still achieve a 30%-53% higher success rate than other agent strategies while maintaining low time and cost. These findings emphasize that the flow-based agent strategy we designed can also maintain a high compilation success rate even under LLMs with different parameter specifications, showing stronger robustness than other agent strategies.

Additionally, combined with the results of the first experiment, we find that the ReAct and Flow-based strategies are more suitable for the compilation task, and the PlanAndExecute strategy appears less suited for the task.

5.4 Ablation Study

In order to evaluate the impact of our designed tools on CompileAgent, we conduct an ablation study. In this experiment, we select GPT-4o with

Table 3: Average tool usage number and ablation result on CompileAgentBench for CompileAgent which is based on GPT-4o.

Tools	**Usage**	**Ablation Result**		
		Csr	$Time$	Exp
CompileAgent	-	89%	8.38	16.53
Shell[1]	-	-	-	-
File Navigator	1.21	81%	6.93	17.32
Instruction Extractor[2]	1.63	77%	7.18	18.26
Website Search	0.61	84%	7.25	16.53
Multi-Agent Discussion	1.87	71%	8.77	18.89

[1] The Shell tool is essential for executing compilation instructions and is a necessary condition for compilation tasks.
[2] We retain the core functionality of the Instruction Extractor while removing the web content crawling feature.

Flow-based as the ablation subject and record the usage frequency of each tool during the compilation process. We then perform the ablation of these tools, and the results are presented in Table 3.

Our experimental results indicate that the Multi-Agent Discussion tool is the most frequently called in the compilation task. Ablating this tool leads to a significant drop in the compilation success rate, reaching 18%, while the time and cost overhead required for compilation also increase. This suggests that CompileAgent relies heavily on the tool when tackling complex problems, as it plays a crucial role in enhancing both accuracy and efficiency. Moreover, the ablation results of the other tools demonstrate their positive contributions to the performance of CompileAgent to varying degrees. Overall, the ablation experiment results confirm the effectiveness and practicality of the tools we designed for real-world compilation tasks.

6 Discussion

6.1 Failure Analysis

In the previous experiments, CompileAgent encounters several compilation failures. After analysis, we summarize the most common three errors in the compilation process: I) Complex Build Dependencies. Some projects rely on intricate dependency chains involving specific versions of libraries, and missing or incompatible dependencies lead to building failures. II) Toolchain Mismatch. Some projects require specific versions of compilers, interpreters, or build tools that are not available or configured properly in the CompileAgent environment, resulting in compilation errors. III) Configuration Complexity. The complex configuration settings in some projects, such as unmatched environmental variables and improperly defined parameters, resulting in the failure of compilation.

6.2 Multi-Language and Multi-Architecture Compilation

Although the CompileAgent in this article is mainly designed for C/C++ projects, it can also support multi-language and multi-architecture compilation due to its scalability and flexibility, and can be expanded to realize the automated compilation process in different environments.

For multi-language compilation, we can first install the interactive environment of each language in Docker and dynamically adjust the toolchain by detecting the programming language used by the project. This includes selecting the appropriate compiler and configuring language-specific build tools, such as javac for Java or npm for JavaScript.

For multi-architecture compilation, we can use the system emulation tools provided by QEMU[5] to enable CompileAgent to interact with environments of different processor architectures such as ARM, MIPS, and X86 to achieve cross-platform compilation.

6.3 Large-Scale Code Analysis

By integrating with multiple code analysis tools, CompileAgent can evaluate the security of repositories during the compilation process, further ensuring the reliability of compilation results, especially for some potentially malicious code repositories. Specifically, we can encapsulate tools such as Coverity Scan[6] and the Scan-Build[7] and call them to perform security analysis when CompileAgent performs compilation, identifying critical vulnerabilities, including buffer overflows or unsafe practices.

7 Conclusion

In this paper, we propose CompileAgent, the first LLM-based agent framework designed for repo-level compilation, which integrates five tools and a flow-based agent strategy to enable LLMs to interact with software artifacts. To assess its performance, we construct a public repo-level compilation benchmark CompileAgentBench, and establish two compilation-friendly schemes as baselines. Experimental results on multiple LLMs demonstrate the effectiveness of CompileAgent. Finally, We also highlight the scalability of CompileAgent and expand its application prospects.

[5] https://www.qemu.org/
[6] https://scan.coverity.com/
[7] https://github.com/llvm/llvm-project

Limitations

Our work is the first attempt to use LLM-based agents to handle the repo-level compilation task, and verify the effectiveness of CompileAgent through comprehensive experiments. However, there are still some limitations that need to be further addressed in the future:

Firstly, CompileAgent relies on the understanding capability of LLMs. During compilation, the agents may misinterpret prompts or instructions, leading to repeated or incorrect actions, which impacts its efficiency in resolving compilation issues. Future work will explore fine-tuning models to improve their in interpreting instructions.

Secondly, the tools incorporated into CompileAgent are relatively basic, leaving unexplored potential for leveraging more advanced programming and debugging tools. Later we can expand the toolset to improve the performance of agents in tackling intricate compilation tasks and error resolution.

Finally, since CompileAgent is highly dependent on the quality of prompt engineering, optimizing the prompts used in the agent system is crucial for its performance. In the future work, we will explore more effective agent strategies to improve overall system performance.

Ethics Consideration

We promise that CompileAgent is inspired by real-world needs for code repositories compilation, with CompileAgentBench constructed from real-world code repositories to ensure practical relevance. During our experiments, all projects were manually reviewed to verify the absence of private information or offensive content. Additionally, we manually compiled each project to validate the reliability of CompileAgentBench.

References

Stanislas G. Bianou and Rodrigue G. Batogna. 2024. Pentest-ai, an llm-powered multi-agents framework for penetration testing automation leveraging mitre attack. In *2024 IEEE International Conference on Cyber Security and Resilience (CSR)*, pages 763–770.

Islem Bouzenia, Premkumar Devanbu, and Michael Pradel. 2024. Repairagent: An autonomous, llm-based agent for program repair. *arXiv preprint arXiv:2403.17134*.

Justin Chen, Swarnadeep Saha, and Mohit Bansal. 2024. ReConcile: Round-table conference improves reasoning via consensus among diverse LLMs. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 7066–7085, Bangkok, Thailand. Association for Computational Linguistics.

Liang Chen, Yichi Zhang, Shuhuai Ren, Haozhe Zhao, Zefan Cai, Yuchi Wang, Peiyi Wang, Tianyu Liu, and Baobao Chang. 2023. Towards end-to-end embodied decision making via multi-modal large language model: Explorations with gpt4-vision and beyond. *arXiv preprint arXiv:2310.02071*.

Claude. 2024. `https://www.anthropic.com/claude/sonnet`.

Antonia Creswell, Murray Shanahan, and Irina Higgins. 2022. Selection-inference: Exploiting large language models for interpretable logical reasoning. *Preprint*, arXiv:2205.09712.

DeepSeek-AI. 2024. Deepseek-v2: A strong, economical, and efficient mixture-of-experts language model. *Preprint*, arXiv:2405.04434.

Gelei Deng, Yi Liu, Víctor Mayoral-Vilches, Peng Liu, Yuekang Li, Yuan Xu, Tianwei Zhang, Yang Liu, Martin Pinzger, and Stefan Rass. 2024. {PentestGPT}: Evaluating and harnessing large language models for automated penetration testing. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 847–864.

Gemini. 2024. `https://deepmind.google/technologies/gemini/flash`.

GPT-4o. 2024. `https://platform.openai.com/docs/models/gpt-4o`.

Dong Huang, Qingwen Bu, Jie M Zhang, Michael Luck, and Heming Cui. 2023. Agentcoder: Multi-agent-based code generation with iterative testing and optimisation. *arXiv preprint arXiv:2312.13010*.

Xiang Huang, Sitao Cheng, Shanshan Huang, Jiayu Shen, Yong Xu, Chaoyun Zhang, and Yuzhong Qu. 2024. QueryAgent: A reliable and efficient reasoning framework with environmental feedback based self-correction. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 5014–5035, Bangkok, Thailand. Association for Computational Linguistics.

Md. Ashraful Islam, Mohammed Eunus Ali, and Md Rizwan Parvez. 2024. MapCoder: Multi-agent code generation for competitive problem solving. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 4912–4944, Bangkok, Thailand. Association for Computational Linguistics.

Ling Jiang, Junwen An, Huihui Huang, Qiyi Tang, Sen Nie, Shi Wu, and Yuqun Zhang. 2024. Binaryai: Binary software composition analysis via intelligent binary source code matching. In *Proceedings of the*

IEEE/ACM 46th International Conference on Software Engineering, ICSE '24, New York, NY, USA. Association for Computing Machinery.

Dongge Liu, Oliver Chang, Jonathan metzman, Martin Sablotny, and Mihai Maruseac. 2024a. OSS-Fuzz-Gen: Automated Fuzz Target Generation.

Yizhou Liu, Pengfei Gao, Xinchen Wang, Jie Liu, Yexuan Shi, Zhao Zhang, and Chao Peng. 2024b. Marscode agent: Ai-native automated bug fixing. *arXiv preprint arXiv:2409.00899*.

Meta-LLaMa. 2024. https://huggingface.co/meta-llama/Llama-3.1-70B.

MistralAI. 2023. https://huggingface.co/mistralai/Mixtral-8x7B-Instruct-v0.1.

OpenAI. 2023. https://openai.com/index/function-calling-and-other-api-updates/.

OpenAI. 2024. https://openai.com/index/new-embedding-models-and-api-updates/.

Chen Qian, Wei Liu, Hongzhang Liu, Nuo Chen, Yufan Dang, Jiahao Li, Cheng Yang, Weize Chen, Yusheng Su, Xin Cong, Juyuan Xu, Dahai Li, Zhiyuan Liu, and Maosong Sun. 2024a. ChatDev: Communicative agents for software development. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 15174–15186, Bangkok, Thailand. Association for Computational Linguistics.

Cheng Qian, Bingxiang He, Zhong Zhuang, Jia Deng, Yujia Qin, Xin Cong, Zhong Zhang, Jie Zhou, Yankai Lin, Zhiyuan Liu, and Maosong Sun. 2024b. Tell me more! towards implicit user intention understanding of language model driven agents. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1088–1113, Bangkok, Thailand. Association for Computational Linguistics.

Xiangmin Shen, Lingzhi Wang, Zhenyuan Li, Yan Chen, Wencheng Zhao, Dawei Sun, Jiashui Wang, and Wei Ruan. 2024. Pentestagent: Incorporating llm agents to automated penetration testing. *arXiv preprint arXiv:2411.05185*.

Cheng Tan, Chenhao Xie, Ang Li, Kevin J. Barker, and Antonino Tumeo. 2020. Opencgra: An open-source unified framework for modeling, testing, and evaluating cgras. In *2020 IEEE 38th International Conference on Computer Design (ICCD)*, pages 381–388.

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

Han Wang, Archiki Prasad, Elias Stengel-Eskin, and Mohit Bansal. 2024a. Soft self-consistency improves language models agents. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pages 287–301, Bangkok, Thailand. Association for Computational Linguistics.

Hao Wang, Zeyu Gao, Chao Zhang, Zihan Sha, Mingyang Sun, Yuchen Zhou, Wenyu Zhu, Wenju Sun, Han Qiu, and Xi Xiao. 2024b. Clap: Learning transferable binary code representations with natural language supervision. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2024, page 503–515, New York, NY, USA. Association for Computing Machinery.

Lei Wang, Chen Ma, Xueyang Feng, Zeyu Zhang, Hao Yang, Jingsen Zhang, Zhiyuan Chen, Jiakai Tang, Xu Chen, Yankai Lin, et al. 2024c. A survey on large language model based autonomous agents. *Frontiers of Computer Science*, 18(6):186345.

Lei Wang, Wanyu Xu, Yihuai Lan, Zhiqiang Hu, Yunshi Lan, Roy Ka-Wei Lee, and Ee-Peng Lim. 2023. Plan-and-solve prompting: Improving zero-shot chain-of-thought reasoning by large language models. *arXiv preprint arXiv:2305.04091*.

Qineng Wang, Zihao Wang, Ying Su, Hanghang Tong, and Yangqiu Song. 2024d. Rethinking the bounds of LLM reasoning: Are multi-agent discussions the key? In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 6106–6131, Bangkok, Thailand. Association for Computational Linguistics.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, Hoang H. Tran, Fuqiang Li, Ren Ma, Mingzhang Zheng, Bill Qian, Yanjun Shao, Niklas Muennighoff, Yizhe Zhang, Binyuan Hui, Junyang Lin, Robert Brennan, Hao Peng, Heng Ji, and Graham Neubig. 2024e. OpenHands: An Open Platform for AI Software Developers as Generalist Agents. *Preprint*, arXiv:2407.16741.

Zhiheng Xi, Wenxiang Chen, Xin Guo, Wei He, Yiwen Ding, Boyang Hong, Ming Zhang, Junzhe Wang, Senjie Jin, Enyu Zhou, et al. 2023a. The rise and potential of large language model based agents: A survey. *arXiv preprint arXiv:2309.07864*.

Zhiheng Xi, Senjie Jin, Yuhao Zhou, Rui Zheng, Songyang Gao, Jia Liu, Tao Gui, Qi Zhang, and Xuanjing Huang. 2023b. Self-Polish: Enhance reasoning in large language models via problem refinement. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pages 11383–11406, Singapore. Association for Computational Linguistics.

Tianbao Xie, Fan Zhou, Zhoujun Cheng, Peng Shi, Luoxuan Weng, Yitao Liu, Toh Jing Hua, Junning Zhao, Qian Liu, Che Liu, Leo Z. Liu, Yiheng Xu, Hongjin Su, Dongchan Shin, Caiming Xiong, and Tao Yu. 2023. Openagents: An open platform for language agents in the wild. *Preprint*, arXiv:2310.10634.

Hanqi Yan, Qinglin Zhu, Xinyu Wang, Lin Gui, and Yulan He. 2024. Mirror: Multiple-perspective self-reflection method for knowledge-rich reasoning. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 7086–7103, Bangkok, Thailand. Association for Computational Linguistics.

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. *Preprint*, arXiv:2405.15793.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2022. React: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629*.

Tong Ye, Lingfei Wu, Tengfei Ma, Xuhong Zhang, Yangkai Du, Peiyu Liu, Shouling Ji, and Wenhai Wang. 2023. CP-BCS: Binary code summarization guided by control flow graph and pseudo code. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 14740–14752, Singapore. Association for Computational Linguistics.

Kechi Zhang, Jia Li, Ge Li, Xianjie Shi, and Zhi Jin. 2024a. CodeAgent: Enhancing code generation with tool-integrated agent systems for real-world repo-level coding challenges. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 13643–13658, Bangkok, Thailand. Association for Computational Linguistics.

Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. 2024b. Autocoderover: Autonomous program improvement. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2024, page 1592–1604, New York, NY, USA. Association for Computing Machinery.

A Benchmark Details

Table 4 presents the composition of CompileAgentBench, which includes 100 popular projects across 14 topics. To align with the distribution of compilation guides in real-world code repositories, CompileAgentBench maintains a ratio of compilation guides in repo to those not in repo, as well as those without guides, at 7:2:1.

B Readme-AI Details

Figure 3 shows the Readme-AI how to be used in our compilation task. Its workflow is that GPT-4o mini first traverses all project files, generate a `Readme.md` file based on specific requirements, and finally MasterAgent can find the compilation instructions by reading the `Readme.md`.

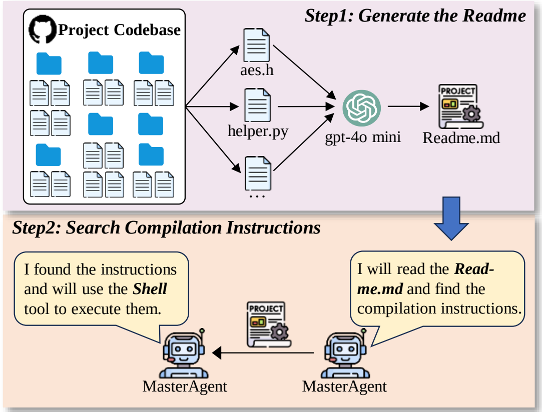

Figure 3: The details of Readme-AI.

C RAG Details

Figure 4 illustrates how the RAG technology is applied in our compilation task. We first specify some files that may contain compilation instructions, such as README, INSTALL, etc., and then split the contents of the files into chunks and generate embeddings and store them in the embedding database. Finally, MasterAgent retrives the embedding database to obtain the compilation instructions. The embedding model we use in this article is text-embedding-3-large (OpenAI, 2024).

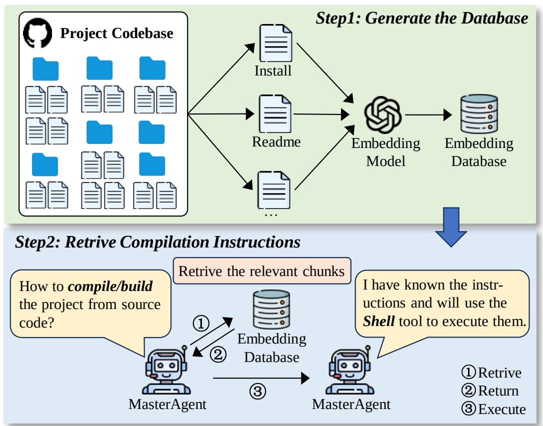

Figure 4: The details of RAG.

Table 4: The composition of CompileAgentBench.

Project	Topic	Existing Guide		No Guide	Project	Topic	Existing Guide		No Guide
		InRepo	*NotInRepo*				*InRepo*	*NotInRepo*	
FFmpeg	Audio	✓	×	×	libvips	Image	✓	×	×
aubio	Audio	✓	×	×	mozjpeg	Image	✓	×	×
cava	Audio	✓	×	×	clib	Linux	✓	×	×
Julius	Audio	✓	×	×	activate-linux	Linux	✓	×	×
zstd	Compression	✓	×	×	libbpf	Linux	✓	×	×
7z	Compression	×	✓	×	util-linux	Linux	✓	×	×
zlib	Compression	×	✓	×	ttygif	Linux	✓	×	×
lz4	Compression	✓	×	×	box64	Linux	✓	×	×
libarchive	Compression	✓	×	×	fsearch	Linux	×	✓	×
mbedtls	Crypto	✓	×	×	uftrace	Linux	✓	×	×
libsodium	Crypto	✓	×	×	libtree	Linux	✓	×	×
wolfssl	Crypto	×	✓	×	toybox	Linux	✓	×	×
nettle	Crypto	×	✓	×	tinyvm	Linux	×	×	✓
libtomcrypt	Crypto	✓	×	×	libpcap	Linux	×	×	✓
libbcrypt	Crypto	✓	×	×	curl	Networking	×	✓	×
tiny-AES-c	Crypto	×	×	✓	masscan	Networking	✓	×	×
boringssl	Crypto	✓	×	×	Mongoose	Networking	×	✓	×
tea-c	Crypto	✓	×	×	libhv	Networking	✓	×	×
cryptopp	Crypto	×	✓	×	wrk	Networking	×	×	✓
botan	Crypto	×	✓	×	dsvpn	Networking	✓	×	×
openssl	Crypto	✓	×	×	streem	Networking	✓	×	×
Tongsuo	Crypto	✓	×	×	vlmcsd	Networking	×	×	✓
GmSSL	Crypto	✓	×	×	acl	Networking	✓	×	×
libgcrypt	Crypto	✓	×	×	odyssey	Networking	✓	×	×
redis	Database	✓	×	×	massdns	Networking	✓	×	×
libbson	Database	×	✓	×	h2o	Networking	×	✓	×
beanstalkd	Database	✓	×	×	ios-webkit-debug-proxy	Networking	✓	×	×
wiredtiger	Database	×	✓	×	whisper.cpp	NN[2]	✓	×	×
sqlite	Database	✓	×	×	llama2.c	NN	✓	×	×
ultrajson	DataProcessing	×	×	✓	pocketsphinx	NN	✓	×	×
webdis	DataProcessing	✓	×	×	lvgl	Programming	×	×	✓
jansson	DataProcessing	✓	×	×	libui	Programming	✓	×	×
json-c	DataProcessing	✓	×	×	quickjs	Programming	×	✓	×
libexpat	DataProcessing	✓	×	×	flex	Programming	✓	×	×
libelf	DataProcessing	×	×	✓	libmodbus	Security	✓	×	×
libusb	Embedded	×	✓	×	msquic	Security	✓	×	×
wasm3	Embedded	✓	×	×	dount	Security	✓	×	×
rtl_433	Embedded	✓	×	×	redsocks	Security	×	✓	×
can-utils	Embedded	✓	×	×	pwnat	Security	×	×	✓
cc65	Embedded	×	✓	×	suricata	Security	×	✓	×
libffi	Embedded	✓	×	×	tini	Security	✓	×	×
uhubctl	Embedded	✓	×	×	tmux	Terminal	✓	×	×
open62541	Embedded	×	✓	×	sc-im	Terminal	✓	×	×
snapraid	Embedded	✓	×	×	pspg	Terminal	✓	×	×
cglm	HPC[1]	✓	×	×	smenu	Terminal	✓	×	×
blis	HPC	✓	×	×	no-more-secrets	Terminal	✓	×	×
zlog	HPC	✓	×	×	linenoise	Terminal	×	×	✓
ompi	HPC	×	✓	×	shc	Terminal	✓	×	×
coz	HPC	✓	×	×	hstr	Terminal	✓	×	×
ImageMagick	Image	×	✓	×	goaccess	Terminal	✓	×	×

[1] HPC stands for High Performance Computing.
[2] NN stands for Neural Network.