CXXCrafter: An LLM-Based Agent for Automated C/C++ Open Source Software Building

ZHENGMIN YU, Fudan University, China
YUAN ZHANG, Fudan University, China
MING WEN, Huazhong University of Science and Technology, China
YINAN NIE, Fudan University, China
WENHUI ZHANG, Fudan University, China
MIN YANG, Fudan University, China

Project building is pivotal to support various program analysis tasks, such as generating intermediate representation code for static analysis and preparing binary code for vulnerability reproduction. However, automating the building process for C/C++ projects is a highly complex endeavor, involving tremendous technical challenges, such as intricate dependency management, diverse build systems, varied toolchains, and multifaceted error handling mechanisms. Consequently, building C/C++ projects often proves to be difficult in practice, hindering the progress of downstream applications. Unfortunately, research on facilitating the building of C/C++ projects remains to be inadequate. The emergence of Large Language Models (LLMs) offers promising solutions to automated software building. Trained on extensive corpora, LLMs can help unify diverse build systems through their comprehension capabilities and address complex errors by leveraging tacit knowledge storage. Moreover, LLM-based agents can be systematically designed to dynamically interact with the environment, effectively managing dynamic building issues. Motivated by these opportunities, we first conduct an empirical study to systematically analyze the current challenges in the C/C++ project building process. Particularly, we observe that most popular C/C++ projects encounter an average of five errors when relying solely on the default build systems. Based on our study, we develop an automated build system called `CXXCrafter` to specifically address the above-mentioned challenges, such as dependency resolution. Our evaluation on open-source software demonstrates that `CXXCrafter` achieves a success rate of 78% in project building. Specifically, among the Top100 dataset, 72 projects are built successfully by both `CXXCrafter` and manual efforts, 3 by `CXXCrafter` only, and 14 manually only. Despite the slightly lower performance, `CXXCrafter` can save tremendous manual efforts and can also be easily applied to a wider range of applications automatically.

CCS Concepts: • **Software and its engineering → Software notations and tools**.

Additional Key Words and Phrases: Agent, Software Building

ACM Reference Format:
Zhengmin Yu, Yuan Zhang, Ming Wen, Yinan Nie, Wenhui Zhang, and Min Yang. 2025. CXXCrafter: An LLM-Based Agent for Automated C/C++ Open Source Software Building. *Proc. ACM Softw. Eng.* 2, FSE, Article FSE116 (July 2025), 23 pages. https://doi.org/10.1145/3729386

Authors' Contact Information: Zhengmin Yu, Fudan University, Shanghai, China, zmyu23@m.fudan.edu.cn; Yuan Zhang, Fudan University, Shanghai, China, yuanxzhang@fudan.edu.cn; Ming Wen, Huazhong University of Science and Technology, Wuhan, China, mwenaa@hust.edu.cn; Yinan Nie, Fudan University, Shanghai, China, 24210240089@m.fudan.edu.cn; Wenhui Zhang, Fudan University, Shanghai, China, 21009100158@stu.xidian.edu.cn; Min Yang, Fudan University, Shanghai, China, m_yang@fudan.edu.cn.

ACM 2994-970X/2025/7-ARTFSE116
https://doi.org/10.1145/3729386

1 Introduction

Open Source Software (OSS) is extensively used across various sectors and plays a critical role in powering modern technological systems, from critical infrastructure to innovative applications. Many foundational software systems, such as operating systems and database management systems, are written in C/C++ [35]. Vulnerabilities in these systems can cause significant damage [47](e.g., the Heartbleed bug in OpenSSL [6]), making automated vulnerability detection for C/C++ OSS essential. Additionally, automated analysis of C/C++ OSS is crucial for areas like vulnerability management, quality assurance, and performance optimization.

Building C/C++ software [53] involves compiling, resolving dependencies, linking libraries, configuring environments, and managing platform-specific challenges. These processes are critical for performing automated program analysis, especially for dynamic analysis. Such tasks require the project to be built into a binary beforehand. However, a significant gap remains in the realm of automated analysis of C/C++ OSS [15]: the absence of a standardized, automated method for building repositories from C/C++ source code. Bridging this gap is essential for enhancing the efficiency and effectiveness of program analyses. The significance of automatically building software from source code for automated analysis can be summarized from two key aspects: (1) **Facilitation of Static Program Analysis**: Many static analysis tasks rely on intermediate representations (IR), such as LLVM IR [29]. This typically requires that the project successfully installs dependencies and can be compiled. (2) **Enablement of Dynamic Program Analysis**: Dynamic program analysis like fuzzing, also requires that the program be compilable from source code, particularly when source-level instrumentation is needed. Automatically built projects can also assist with several downstream tasks, such as automated vulnerability reproduction tasks.

Existing research efforts [16, 18, 20, 36, 59, 60] mainly focus on Java/Python, while C/C++ remains underexplored due to its higher complexity. Unlike the relatively unified and automated build and package management tools in Java (e.g., *Maven* [2], *Gradle* [13]) and JavaScript (e.g., *NPM* [38]), or Python's convenient *pip* [48], the C/C++ ecosystem contains over 20 distinct build systems [7], with lower levels of standardization and automation, posing significant challenges.

To better understand the automation of C/C++ project builds, we investigate the build systems of 100 popular open-source C/C++ projects across 10 different categories, using their default build commands and settings. The study shows that more than 70% of these projects fail to be built successfully without manual intervention, suggesting that most C/C++ projects require additional configuration, such as downloading dependencies or setting compilation parameters. To further investigate the root causes of these failures, we manually fix the errors encountered during the build process guided by the failure messages iteratively until successful completion. In total, we encounter 384 errors across 79 projects, and spend over 153 man-hours to resolve them. This underscores the significant challenges in automating the C/C++ build process.

Challenges. Drawing from the root causes and insights gathered in our study, we summarize the following challenges associated with C/C++ build automation:

- **Challenge 1: Complexity of Dependency Management.** C/C++ projects often rely on substantial external libraries and tools, which require careful management and configuration of dependencies. Although package management tools like *Conan* [23] and *vcpkg* [33] are available, they support different sets of libraries and have distinct usage patterns, which makes dependency management a complex task.
- **Challenge 2: Diversity of Build Systems and Compilation Options.** C/C++ projects adopt at least 20 different build systems (such as *Makefile* [10], *CMake* [22], *Autotools* [11], and *SCons* [52]), each with unique syntax and configuration requirements. Additionally, these projects employ a wide array of compilers (such as *GCC* [9] and *Clang* [39]) and

toolchains, each with its own options and configuration methods. Such diverse build systems and toolchains would trigger substantial errors when building large-scale real projects.
- **Challenge 3: Complexity of Error Diagnosis and Debugging.** The diverse build process in C/C++ projects often generate many error messages at multiple levels, such as pre-processing, compilation, and linking, which often vary greatly across different projects.

Our System. Large Language Models (LLMs) are renowned for their strong capabilities in understanding complex documentations [19, 62], generating structured instructions [24, 50], and resolving errors [14, 49]. Inspired by such abilities, we investigate whether LLMs can extend their effectiveness to the domain of build systems and error resolution. In particular, we apply LLMs (*i.e., GPT4o*) to solve specific build issues identified in our empirical study and observe that it can successfully address several of them (as demonstrated in the experiment in Section 5.1). For instance, during *GCC* builds, the LLM can automatically install dependencies like *GMP*, *MPFR*, *MPC*, and *Flex*, and prompts 64-bit compilation, thus avoiding errors from the default build instructions and simplifying dependency management. Such results demonstrate LLM's potential capability in this domain. At the same time, it also indicates that for complex project builds, which involve multi-step processes, the effectiveness of standalone LLMs is limited. Relying solely on LLMs can only address a small fraction of errors, highlighting the need for more refined strategies capable of continuously addressing build failures.

To address the above challenges, we propose an LLM-based agent system named `CXXCrafter` that leverages LLMs to dynamically manage complex build processes. The system consists of three modules: the **Parser Module**, the **Generator Module**, and the **Executor Module**. Specifically, the **Parser Module** automatically extracts and parses relevant information from the repositories, such as dependencies and build system configurations. The **Generator Module** utilizes LLMs to generate candidate build solutions (i.e., Dockerfiles, which include shell scripts for the entire software build process) based on the parsed information. Additionally, the Generator is responsible for modifying the candidate build solutions in response to error feedback from the Executor. The **Executor Module** oversees the build process in the Docker container where the build is performed, capturing error messages and determining whether the build is successful. The Generator and Executor form a dynamic interaction loop, continuously addressing build issues until the process completes successfully. Our design effectively addresses the three challenges as mentioned above. In particular, the Parser can identify the required dependencies to avoid potential dependency errors. Besides, `CXXCrafter` also employs an automated, iterative feedback process powered by LLMs to dynamically identify and install dependencies, thus effectively addressing issues such as uncertain dependencies or version conflicts (Challenge 1). Furthermore, `CXXCrafter` leverages LLMs' rich domain knowledge via nested prompt templates to unify different build systems and compilation options (Challenge 2). For the Challenge 3, `CXXCrafter` captures real-time feedback during the build process, enabling efficient error diagnosis and debugging by adapting to both known and new errors arising during the build.

We evaluate `CXXCrafter` on both the aforementioned 100 popular C/C++ projects and the larger Awesome-CPP dataset [7], which includes 652 projects across various categories. Specifically, `CXXCrafter` successfully builds 587 out of the 752 projects, achieving a success rate of 78%. This significantly outperforms other heuristic approaches (39.01%) and bare LLM (34.22%). Though its overall performance does not surpass the build success rate achieved by humans, `CXXCrafter` resolves three projects that cannot be successfully built through human efforts. Our analysis for these three projects shows that `CXXCrafter` leverages the implicit build knowledge embedded in the LLM and the powerful retrieval capabilities of its parser module, offering unique advantages even compared to human efforts in project builds. Additionally, a component analysis demonstrates its

effectiveness in designing agents capable of handling complex tasks. Finally, we assess the efficiency and cost, and the results show that it takes 875 seconds together with a financial cost of $0.41 per successful build. These evaluation experiments underscore the practical value of `CXXCrafter`.

Contributions. This paper makes the following main contributions:

- **Originality:** To our best knowledge, we are the first to explore the idea of utilizing LLM agent to automate C/C++ build process, and our study demonstrates promising results.
- **Empirical Study:** We conduct an empirical study on the build processes of 100 popular open-source C/C++ projects to understand the current state of build tools. By identifying and categorizing 384 build errors, we provide a comprehensive analysis of the challenges to automate C/C++ builds, offering key findings to the root causes of such failures.
- **Approach:** We propose `CXXCrafter`, an LLM-based agent system designed to automate the build process for large-scale C/C++ repositories. In particular, `CXXCrafter` dynamically manages dependencies, resolves build issues, and diagnoses errors, effectively addressing the challenges such as handling various build systems and installing dependencies.
- **Evaluation:** Through extensive evaluations on 752 projects, `CXXCrafter` achieves an impressive build success rate of 78%, demonstrating its pioneering effectiveness in C/C++ build automation. Our research has the potential to support downstream program analysis efforts.

2 Background & Related Work

Software Building. Software building [53] converts code into executables or libraries, involving tasks like dependency resolution, compilation, and linking. For large projects, automated build systems become essential, as manual handling becomes impractical. These systems streamline the process, managing tasks efficiently. Different programming languages have specific build systems: Java uses *Apache Ant* [1], *Maven* [2], and *Gradle*, while JavaScript relies on *NPM* [38], and Python uses *setuptools* [40]. In C/C++ projects, tools like *CMake*, *Make*, *Ninja*, and *Bazel* are frequently used. Additionally, building differs from compiling, which is just one part of the broader building process, and Continuous Integration (CI), where building is a prerequisite for integration.

Several studies focus on automating software builds, mostly for languages like Java, with fewer addressing C/C++ projects. Hassan et al. [16] investigate Java build failures, revealing that 86 out of 200 projects fail to build automatically using default commands. Other studies have explored build [18, 36, 59, 60] and CI failures [58]. For example, Lou et al. [30] analyzed 1,080 build issues from Stack Overflow related to *Maven*, *Ant*, and *Gradle*, finding that 67.96% of the issues were resolved by modifying build scripts for plugins and dependencies. Similarly, Olivier et al. [37] analyzed over 1.2 million build logs from Google's *OSS-Fuzz* service to identify common failure patterns. In the context of C/C++, we only found *CPPBuild* [15] for automating the build process. But it is limited to *CMake*, *Make*, and *Autotools*, resulting in lower accuracy for open-source projects with other build systems. Furthermore, while some works focus on containerization techniques and Dockerfile generation [17, 32, 43], they typically do not address building software from source.

LLMs and Agents. LLMs have shown outstanding performance across multiple dimensions, including semantic understanding [46], code generation [21], and implicit knowledge storage [57]. However, they still face several limitations [28, 42], such as solving complex tasks, maintaining context over long interactions, executing actions in real-world environments, and engaging in dynamic, multi-turn dialogues. LLM-based agents, designed to address these challenges, integrate more advanced functionalities. They are increasingly used in a variety of scenarios [3], including code generation [12, 51] and security tasks [5], showing significant promise for future advancements.

Table 1. The Top 100 Projects and Their Categories

Field	Projects
Kernel and System Programming Tools	GCC, 8CC, Codon, Mold, Clang, LLVM-project, Gdb, Linux, Reactos, RT-Thread
Game Development	Godot, GamePlay, raylib, DOOM, Entt, Stockfish, Cocos2d-x, Rpcs3, OpenRCT2, Minetest
Image Processing	Guetzli, Openalpr, Openpose, Aseprite, Tesseract, Mozjpeg, Libfacedetection, Libjpeg-turbo, Libjxl, Simd
AI Frameworks and Tools	Tensorflow, Mediapipe, LocalAI, XGBoost, OpenCV, GPT4All, Llama.cpp, Caffe, Paddle, Mxnet
Database Development	Rethinkdb, Mongo, Leveldb, Rocksdb, Sqlitebrowser, Duckdb, Foundationdb, Arangodb, Scylladb, Kvrocks
Video Processing	Srs, FFmpeg, Libvpx, Openh264, Vireo, Theora, X265, Blender, Shotcut, Libde265
Audio Systems	Sonic-pi, OpenFrameworks, Rnnoise, Soloud, Aubio, Libsndfile, MuseScore, Wav2letter, Libsoundio, AudioFlux
Web Frameworks	Treefrog-framework, Civetweb, Pistache, Cpprestsdk, Crow, Drogon, Facil.io, Lwan, Oatpp, Userver
GUI	NanoGUI, RmlUi, Elements, Libui, Xtd, Flameshot, Qv2ray, Polybar, DearPyGui, Webui
IDEs	Geany, Code::Blocks, Jucipp, Color_coded, CodeLite, rtags, serenity, qt-creator, cquery, Irony-mode

3 Empirical Study

In this section, we conduct an empirical study to assess the current status of building C/C++ projects, aiming to determine how effectively existing build systems can handle the complexities and challenges of real-world projects. Our research team consists of 4 programmers, each with extensive experience in C/C++ development and building. Specifically, we manually attempt to build 100 widely-used C/C++ projects, devoting approximately 153 man-hours to resolving the generated build failures. Out of the 100 projects, 86 have been built successfully, while the remaining projects either require excessive time budget or encounter unresolved issues. Additionally, we analyze the errors encountered during the building process and summarize the root causes. The research questions, datasets, and study results are presented in detail as follows.

Research Questions. Referring to a recent study [16], which investigates the build mechanism and ecosystem of Java, we design the following research questions for the study on C/C++ OSS:

- **RQ1 (Default Build Success Rate):** What proportion of popular C/C++ projects can be successfully built using their respective build systems and default build commands?
- **RQ2 (Build Failure Causes):** What are the major root causes of the observed build failures among these projects?

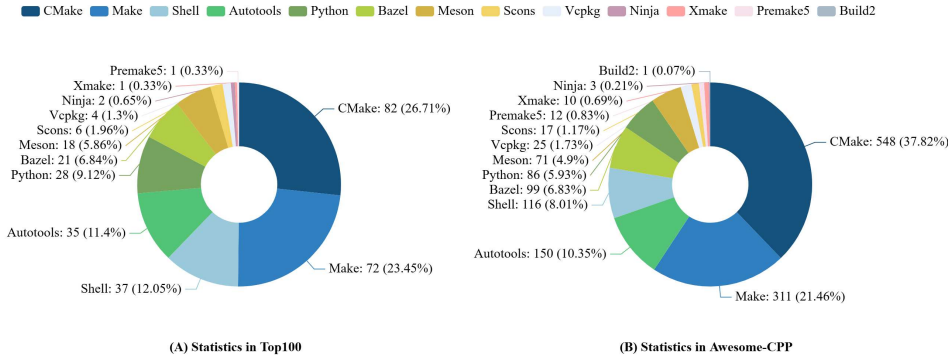

Fig. 1. The Statistics of Build Tools used in the Top 100 and Awesome-CPP Datasets (introduced in Section 5).

Dataset. For our empirical study, we construct a dataset (hereinafter referred to as Top100) via selecting the top 100 most popular open-source C/C++ projects from GitHub, spanning 10 distinct categories to ensure diversity and comprehensiveness. These categories include foundational projects such as operating systems, database management systems, as well as emerging projects like AI frameworks. The projects, as summarized in Table 1, are mostly the top 10 in their respective

fields based on star ratings, except for those that do not meet the following requirements. Since our builds are conducted on a Linux system, we exclude any projects that are incompatible with Linux builds (e.g., *CnC_Remastered_Collection*). Additionally, repositories that are not fully open-source (e.g., *AppCode*, *Cvelop*) or do not qualify as complete projects (e.g., *3d-game-shaders-for-beginners*, *minimp3*) are also excluded. We focus on these repositories because they are frequently analyzed and studied in downstream applications such as program analysis, making them ideal candidates for our research. Additionally, as popular projects, they exemplify common practices and challenges in building C/C++ projects within the open-source community.

Table 2. Results of Executing Default Build Commands on the Top100 Dataset

Category	#Success	Successfully Built Projects
Kernel and System Programming Tools	2	8CC (Make), Mold (CMake)
Game Development	1	Entt (CMake)
Image Processing	5	Guetzli (Bazel), Libfacedetection (CMake), Libjpeg-turbo (CMake), Libjxl (CMake), Simd (CMake)
AI Frameworks and Tools	3	XGboost (CMake), OpenCV (CMake), Llama.cpp (CMake)
Database Development	3	Leveldb (CMake), Duckdb (CMake), Kvrocks (CMake)
Video Processing	1	X265 (CMake)
Audio Systems	2	Libsndfile (CMake), Libsoundio (CMake)
Web Application Frameworks	3	Civetweb (Make), Facil.io (CMake), Oatpp (CMake)
GUI	1	Webui (CMake)
IDE	0	None
Sum	21	N/A

3.1 Success Rate of Default Build Commands (RQ1)

To answer **RQ1**, we employ a three-phase process to apply default build commands to each C/C++ project. In the **first phase**, we gather the most commonly used build commands for popular build tools through an extensive review of online tutorials and documentation. For example, we choose '*make*' for Makefile-based projects, '*mkdir build && cmake .. && make*' for *CMake* combined with *make*, and '*./configure && make*' for *Autotools*. A complete list of build systems and their corresponding default commands is provided in the appendix [56]. In the **second phase**, we select the build systems used by the 100 projects. We manually inspect the project's source code directory to identify the build system. For projects that support multiple build systems, we determine the primary system and entry files (the files used to initiate the build process) based on the official documentation. If the documentation does not offer a clear recommendation, we randomly select one to proceed with. In cases where the selected build system fails in the subsequent steps, we switch to another one. If the chosen system succeeds, the process completes. In the **third phase**, we apply the appropriate build commands to each project. To ensure consistency, all builds are executed separately within a newly installed *Ubuntu 22.04* Docker environment, without any pre-installed dependencies. If a project has specific OS requirements, we switch to the required system.

During the build process, we document the build systems used in the 100 projects, as shown in Figure 1. The statistics reveal significant variability in the build systems employed by popular projects. In particular, most projects support *CMake* and *Make*, with these two systems often being used in combination. The results of applying the default build commands to the Top100 dataset are presented in Table 2. As shown in the table, only 21 projects are successfully built, highlighting that even well-known and actively maintained projects demonstrate low compatibility with default configurations. For the remaining 79 projects, we observe the failure reasons can be attributed to a lack of specific setups, which can be mainly categorized into 3 types. **First**, 51 projects encounter dependency-related errors, where required dependencies, such as *libpng* when building *mozjpeg*, are missing and not automatically installed. For projects with missing dependencies, we manually review the project's documentation, including files like "README", "Contribution", "Compile",

and "Building", to check for any information on dependencies required before building. Out of the 51 projects, 28 have missing dependencies that are not mentioned in their documentation. Many projects do not clearly specify which dependencies are required, forcing developers to spend extra time addressing these issues. **Second**, 17 projects face issues related to incompatible build system versions or missing tools. For example, the *bazel* version required for building *mediapipe* does not meet the requirements. **Third**, 11 projects fail due to incorrect build commands, such as needing to specify the target as '*build*' when running '*make*' for *LocalAI*. In total, resolving these issues for the 79 failed projects requires additional, non-default configurations across all three categories.

Finding1: The build systems of C/C++ projects vary significantly, yet the level of automation among existing systems remains relatively low. Furthermore, many projects often require additional specific setup steps to build successfully.

Table 3. Results of the Build Process by Humans on the Top100 Dataset

Category	#Success	Not Successfully Built Projects
Kernel and System Programming Tools	8	Codon, RT-Thread
Game Development	8	DOOM, Cocos2d-x
Image Processing	10	None
AI Frameworks and Tools	8	LocalAI, Paddle
Database Development	7	FoundationDB, ArangoDB, Scylladb
Video Processing	10	None
Audio Systems	7	OpenFramework, MuseScore, Wav2letter
Web Frameworks	10	None
GUI	9	Qv2ray
IDEs	9	Cquery
Sum	86	N/A

3.2 Root Causes of Build Failures in Actual Build Processes (RQ2)

To answer **RQ2**, we continue building the 79 C/C++ projects that initially failed with the default build commands by systematically investigating each build failure. Leveraging expert knowledge and online resources, 4 programmers resolve errors one by one, documenting each issue and verifying the resolution by ensuring the error no longer occurs. In total, we have successfully built 65 out of the 79 projects. However, 14 projects cannot be built for two reasons: *unresolved source code errors* and *exceeding the four-hour build time limit*. Among these 14 projects, 9 encounter errors that cannot be resolved (e.g., the "Unknown *CMake* command '*harfbuzz_Populate*' " in *MuseScore*). We determine that these errors are unlikely to be fixed because similar issues have been reported by other developers in the official GitHub repositories, yet the project maintainers have not provided effective solutions. By searching the official GitHub issues using keywords from the error messages, we find that, of the 9 issues, 7 are still open and 2 are closed. However, even for the closed issues, the proposed solutions do not resolve our build problems. Additionally, 5 out of 14 projects fail due to timeouts. Based on our observation, projects that exceed 4 hours typically do not resolve independently. Compiling large projects, such as the *Linux kernel*, takes less than 20 minutes on our server, and the four-hour window allows sufficient attempts to address any issues. Therefore, we consider projects that exceed this time limit as failures to avoid unnecessary time expenditure.

After completing all the build processes, we resolve a total of 384 errors, nearly 5 errors per project on average. By conducting a systematic taxonomy, we categorize the root causes of these failures, as summarized in Table 4. The build failures of C/C++ projects are classified into three main categories: library issues, build toolchain issues, and configuration issues. In addition to these, we also identified other factors that contributed to the failures, such as code errors within the projects, which are classified as other issues. The following introduces the details.

Table 4. Root Causes of Build Errors in the Building of the Top 100 Projects by Humans

Category	Subcategory	Count	Description
Library Issues	Library Not Installed	263	Required libraries are not installed, resulting in missing dependencies during the build.
	Library Not in Path	10	Libraries exist but are not included in the build system's paths such as '*LD_LIBRARY_PATH*', so the system is not able to locate them.
	Library Version Inconsistency	11	The installed library version is inconsistent with what the project requires.
	Sum	284	
Build Toolchain Issues	Build System Version Conflict	6	The build system, such as *CMake* or *Make*, is incompatible or outdated, causing build failures.
	Other Tools Missing or Conflicting	58	Build tools, excluding the build system, such as compilers and package managers like *pkgconfig* and *vcpkg*, are either missing or have version conflicts.
	Sum	64	
Configuration Issues	OS/Platform Incompatibility	7	Incompatibilities arising from the operating system or hardware architecture, including discrepancies in system versions or hardware support for libraries such as *CUDA* and *cuDNN*.
	Incorrect Build Commands	10	Non-standard or incorrect build commands, including missing flags or options.
	Project Configuration Issues	13	The lack of project-specific configurations, such as a defined file structure, initialization scripts, or dependency installation scripts.
	Sum	30	
Other Issues	Memory Issues	1	Errors due to insufficient memory or memory allocation problems during the build process.
	Source Code Issues	3	Errors in the source code, such as syntax errors, undefined references, or incorrect logic.
	Unstable Builds in Certain Branches	2	Instability in certain project branches, resulting in inconsistent build outcomes.
	Sum	6	
Total		384	

3.2.1 Library Issues. Library issues often occur when the required libraries are either not installed, not placed in system environment paths, or have incompatible versions. These issues typically result in errors such as "library not found" or "undefined reference" as the compiler or linker is unable to resolve the symbols or functions defined in those libraries. Among open-source projects, developers often share only the core source code and exclude the installed libraries to keep the repository concise. However, this can lead to library-related errors when others attempt to build the project without the necessary dependencies installed. This issue occurs a total of 284 times in our study, making it the most frequent problem encountered during C/C++ project builds. Compared to other build systems, such as *Maven* or *Gradle* in Java [16], we find that C/C++ build systems generally make less effort to automatically reinstall removed libraries. To some extent, this may be due to the more complex nature of C/C++ dependencies and the lack of a unified package management tool like those found in higher-level languages such as Java or Python. These issues can be further categorized into three sub-categories as follows.

Library Not Installed. In our empirical study, most library issues are attributed to missing libraries, with 263 out of 284 cases falling into this category. These missing libraries are typically, though not always, removed from open-source projects by developers to save space or for other reasons. As a result, builders manually download them through methods such as using package managers or building from source. For instance, during the build of *libde265*, *OpenRCT2*, and *minetest*, *SDL2* is not found and needs to be installed using *apt*.

Errors related to missing libraries frequently occur during the preparation phase when build systems checking for dependencies. However, if left unresolved, they can also surface later during the compilation or linking phases, as observed in the building processes of projects like *rpcs3*, *aseprite*, and *mxnet*. While package management tools like vcpkg and Conan exist for C/C++ development, they are not as widely adopted or standardized as those used in higher-level languages like Java.

Library Not in Path. This issue arises when libraries are installed but not included in the system's search paths, such as '*LD_LIBRARY_PATH*', preventing the build system from locating them. In our study, this occurs 10 times, causing errors during compilation or linking phase when dependencies cannot be resolved. For example, *MuseScore* fails to build because the file '*FindQt6Qml.cmake*' is not found in '*CMAKE_MODULE_PATH*'.

Library Version Inconsistency. This issue occurs when the installed library version is inconsistent with what the project requires. Due to API or behavioral discrepancies, this leads to incompatibilities during the dependency management, linking, or compilation phases. In our study, this issue is observed 11 times in projects such as *Shotcut*, *OpenPose*, and *Sonic-Pi*, where these conflicts resulted in build failures. Resolving such issues typically involves updating the project to accommodate the installed library version or reverting to an older, compatible version.

3.2.2 Build Toolchain Issues. Build toolchain issues refer to problems related to missing or incompatible versions of tools necessary for the build process, such as compilers, linkers, or other essential utilities. These issues typically arise when the project's toolchain is not fully specified or when the available version does not meet the project's requirements. In our study, this occurs 64 times. These toolchain issues can be further divided into two categories, as outlined below.

Build System Version Conflict. This sub-category error occurs 6 times in our study. We ensure that the corresponding build systems are installed by default, thus avoiding missing toolchain issues. However, version mismatches occasionally occur. For example, in the *wav2letter* project, the environment requires a minimum *CMake* version of 3.29.2, but the version available in the default *APT* repositories is 3.25.1. Due to the unique role of build systems within the toolchain, we classify this type of error as a separate sub-category.

Other External Tools Missing or Conflicting. The toolchain also includes external utilities such as debuggers, linker and profilers, which may be necessary for certain stages of the build or testing process. Incompatible or missing versions of these tools caused issues in 58 cases. For example, missing or conflicting versions of utilities like *GDB* or *Valgrind* can lead to failures during debugging or performance analysis stages.

Finding 2: Library issues (e.g., library not installed, version inconsistency) are the most significant challenges in C/C++ project building, followed by build toolchain issues and configuration issues.

3.2.3 Configuration Issues. Configuration issues occur when a project's build scripts are misconfigured or incompatible with the specific environment. These issues include platform or operating system incompatibilities, incorrect build options, and misconfigured files.

System or Equipment Incompatibility. Certain projects are designed to run exclusively on specific operating systems or hardware platforms, and attempting to build them on an unsupported platform often results in failures. For example, projects like *OpenPose* recommend using Ubuntu versions between 14 and 20, while older projects such as *OpenAPLR* suggest *Ubuntu 16.04*. Additionally, hardware-specific requirements, such as the absence of a GPU, can prevent the building of projects reliant on *CUDA* and *cuDNN*. In our evaluation, such errors occurred 7 times.

Incorrect Build Commands Build instructions often require specific setups, such as configuring environment variables, cross-compilation, or managing dependencies. For example, when building for a different architecture like ARM, a toolchain file must be specified to ensure proper compilation: '*cmake -DCMAKE_TOOLCHAIN_FILE=path/to/arm_toolchain.cmake ..*'. Without such configurations, the build process may fail or produce incorrect results.

Project Configuration Issues This error occurred 13 times and is typically caused by missing project-specific configurations, such as hardcoded paths or dependencies hosted on private sources.

For example, the *gameplay* project requires files (e.g., *gameplay-deps*) from a specific URL. Without performing these required custom setups, the build process is bound to fail.

While we also encountered issues such as source code errors and unstable versions, which occurred 6 times. Our study focuses primarily on build system-related problems. We have documented these issues as they pose significant barriers to successful builds.

Finding 3: Build errors in C/C++ projects can occur at various stages, including dependency resolution, compilation, linking, or runtime setup. These issues are diverse in nature, as they vary depending on the build tools and project characteristics involved at each stage.

Building C/C++ projects is challenging, even for human developers, and automating this process adds further complexity. Based on our empirical study, we summarize the key challenges in automating C/C++ builds. Dependency management is a frequent challenge during dynamic builds, involving the identification, downloading, and resolution of issues. While Software Composition Analysis (SCA) studies [36, 54, 55] address dependency issues, they fail to detect non-local third-party libraries (TPLs) before building. Research like *CCScanner* [45] examines package management tools, but build-specific issues like alias conflicts and version mismatches remain unaddressed. Static analysis is insufficient for dependencies that are conditional, dynamically loaded, or tied to build environments with varying compiler flags and OS requirements. Additionally, dependencies often originate from multiple sources, such as package managers (e.g., *apt*) or source code, and the obstacles in downloading them further complicate the resolution process. Second, in our study, we manually write extensive shell scripts and perform debugging within Dockerfiles, utilizing various tools like build systems, compilers, and package managers. The diversity of these tools and their commands makes it difficult to standardize the build process with fixed rules, presenting a major challenge in automation. Lastly, build errors can occur at any stage, including preprocessing, compilation, linking, or even due to external factors like network issues or hardware limitations, such as insufficient memory, can also cause build failures. These errors vary significantly, making it difficult to apply generalized error-handling strategies. Furthermore, the solutions to these problems are often scattered across various sources, requiring extensive expertise or the ability to conduct in-depth research through documentation, community forums, and other resources. All these challenges hinder the automation of building C/C++ projects.

4 Methodology

In light of the challenges discussed in Section 3, we design an agent `CXXCrafter` to streamline the building of C/C++ projects by leveraging LLMs to handle various stages of the building process.

4.1 Overview

Our approach is driven by the broad capabilities of LLMs across multiple dimensions, including semantic understanding [46], code generation [21], and implicit knowledge storage [57]. Existing studies show that LLMs' semantic understanding enables the performance of code analysis tasks [8] and assists in bug and error comprehension [25, 27], demonstrating potential for interpreting diverse error messages in the build process. Their robust code generation capabilities allow developers to create applications in various programming languages [61], with promising potential for automatically generating build instructions and bash scripts to resolve build errors. Additionally, through training on extensive corpora, LLMs implicitly store vast amounts of knowledge across multiple domains, helping to address issues in various fields [31, 41, 44]. LLMs may have been trained on large-scale open-source resources, including GitHub issues and Stack Overflow [4, 26, 34],

which contain numerous build-related problems and solutions, further underscoring their potential in tackling challenges related to software construction.

However, the effectiveness of directly using LLMs for building is limited. As shown in the experimental results in Section 5, using bare LLMs with prompts successfully generates build solutions for only about 30% of the projects. This is because the build process for many C/C++ projects involves multi-faceted errors, including those arising from different stages of the build. Relying solely on the LLM can only address a small fraction of such errors. An iterative approach is needed to continuously resolve issues as they arise. To address this, we propose an LLM-based agent that dynamically manages the build process through iterative feedback mechanisms. This agent autonomously resolves errors in real-time, adjusting and refining build decisions based on evolving conditions. The framework not only reduces the need for manual intervention but also enhances build reliability and success rates.

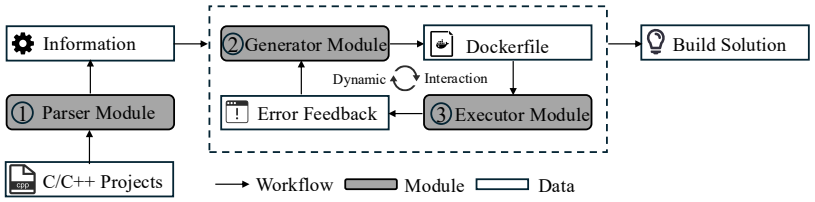

Fig. 2. The Overall Framework of CXXCrafter

As illustrated in Figure 2, the CXXCrafter is comprised of three essential modules:

- **Parser Module**: This module automatically extracts and analyzes key build-related information from the project directory, encompassing dependencies, environment settings, and relevant documentation that facilitate the build process. This ensures that all essential data is available for the subsequent stages of the workflow. Additionally, we leverage the LLM's semantic understanding capabilities to overcome two key obstacles: identifying the valid build system entry file and retrieving helpful documentation.
- **Generator Module**: This module utilizes LLMs to generate a Dockerfile that includes build procedure code based on the parsed information, ensuring that necessary dependencies, environment settings, and configurations are correctly specified. The module also modifies the Dockerfile in response to error feedback from the Executor Module, ensuring an adaptive approach to resolving build issues.
- **Executor Module**: This module oversees the build process in containers by execute Dockerfile, providing a consistent and clean build environment for testing whether the build solution succeeds. Specifically, it captures errors and logs, feeding them back to the Generator Module, forming a dynamic interaction loop that continuously addresses errors until completion.

CXXCrafter uses five types of prompts for different use cases, incorporating techniques such as RAG and nested prompt templates, as detailed in Section 4.5.

4.2 Parser Module

The Parser Module analyzes local projects to extract critical information for the software's environment preparation and compilation.

It employs three specialized extractors (see Figure 3) to gather data, including environment settings, dependency details, and helpful build documentation. ① In the **Environment Information Extractor**, CXXCrafter uses basic shell commands like '*lscpu*' and '*uname -a*' to capture system

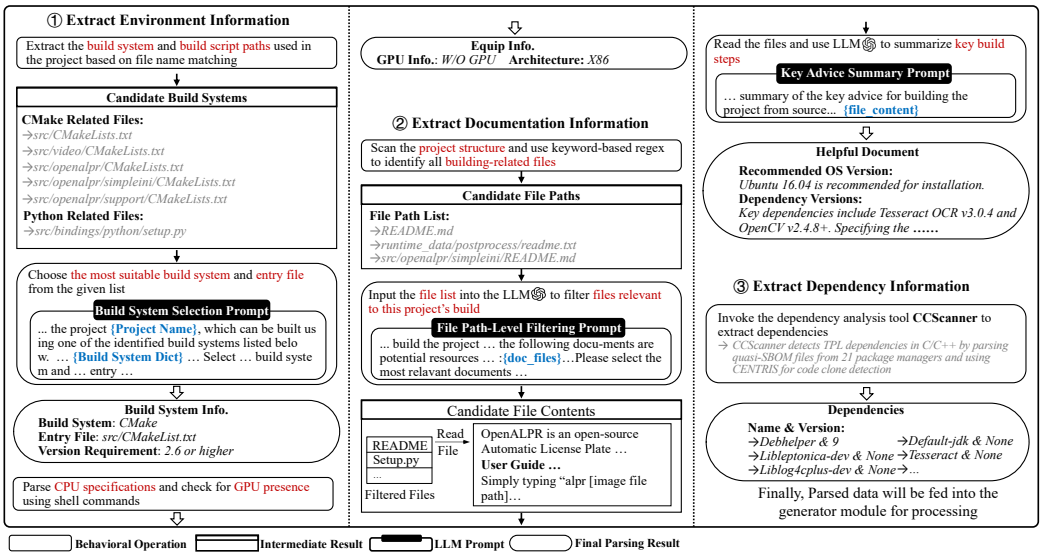

Fig. 3. The Parser Module. It is responsible for automatically extracting and analyzing information such as dependencies, build system information, which is crucial for subsequent build processes.

details, including CPU specs, OS, and their versions. This information is crucial for addressing issues discussed in Section 3.2.3, such as installing software for specific architectures or ensuring the correct GPU/CPU driver versions. It plays a key role in ensuring compatibility and optimization. ② The **Dependency Information Extractor** takes the entire source code folder as input and outputs the names and versions of all required dependencies, helping to prevent conflicts and ensure software stability. Existing research on dependency identification falls into two categories: some studies [36, 54, 55] use Software Composition Analysis (SCA), but SCA can't recognize third-party libraries (TPLs) before build time, as many TPLs are not available locally at that stage. *CCScanner* [45] detects TPL dependencies in C/C++ by parsing quasi-SBOM files from 21 package managers and using *CENTRIS* [54] for code clone detection. We use it in our parser module to statically extract dependency names and versions. It is worth noting that while the statically extracted dependencies help address library-related issues, they do not fully resolve them. Specifically, they are incapable of handling dynamic errors such as aliasing (mismatched resolved and downloaded library names) or version conflicts, which only manifest during the execution process. Static dependency analysis at this stage is insufficient, necessitating the use of the generator and executor modules for dynamic resolution. ③ The **Useful Documentation Extractor** collects relevant build instructions and configuration guides, aiding `CXXCrafter` in troubleshooting and understanding the build process. As shown in Part 2 of Figure 3, it scans the source code folder and applies two rounds of filtering. First, it uses keyword-based regular expressions to identify build-related files and remove irrelevant ones. Then, it performs finer filtering using LLMs, based on the project name and document path, to exclude unrelated files. Finally, it reads the filtered files and uses the LLM to summarize key build information, ultimately obtaining the relevant documents for the build process.

The parser module faces 2 key obstacles: identifying the correct build system and entry file, as well as retrieving useful documentation for the build process. First, many projects employ multiple build systems, each with several build files. Expert knowledge is required to determine which build system and entry file are suited to compile the entire project. We address this by leveraging LLMs

combined with tailored prompts. For example, in the *OpenALPR* project (Figure 3 Part 1), both *CMake* and Python are present, but the LLM correctly identifies *CMake*, recognizing the Python paths as interface files rather than the main project. Second, some projects include useful documentation that aids the build process, but traditional rule-based methods struggle to locate this information. To address this, we develop a RAG system to search for relevant content. For example, in Figure 3, we retrieved documentation from the "README.md" file, which recommended installing *Ubuntu 16.04* and provided advice on dependency versions to help avoid potential compatibility issues.

4.3 Generator Module

The generator module is responsible for creating and modifying build solutions. In `CXXCrafter`, build solutions are defined using Dockerfiles, enabling the construction of C/C++ software in clean and reproducible environments. While Shell or Python scripts can also be used, Docker often offers higher flexibility and consistency. Its ability to generate clean system images ensures that the resulting Dockerfiles can be executed reliably across different environments.

Upon receiving the output from the parser, the generator produces an initial version of the Dockerfile. We have designed curated Embedded Prompt Templates (detailed in Section 4.5), which provide structured guidance to the LLM by embedding predefined formats and placeholders within the prompts. These templates ensure the Dockerfile creation process is structured and consistent. The generator begins modifying the Dockerfile when the executor encounters a failure, utilizing the error message and the recently executed Dockerfile. We retain all modification history within the same session of the LLM and prioritize clearing the oldest resolved issues when the context limit is reached, allowing the model to reference recent decisions during the modification process. Our key methodology in the generator module is the design of Embedded Prompt Templates. These templates offer structured guidance to the LLM by embedding predefined formats and placeholders within the prompts. Drawing from the building experience in Section 3, we have systematically outlined the structure of Dockerfiles in the prompt, encompassing essential components such as system and tool installation, package management updates, dependency installation, project-specific configurations, and build-related instructions. This structured approach ensures consistency and adherence to best practices, promoting the generation of standardized yet flexible build solutions.

4.4 Executor Module

The executor module is responsible for executing the Dockerfile generated by the generator module. It monitors the entire build process to detect errors. During the building process, the executor tracks the executed commands and logs detailed traces. If the build fails, the executor sends the error messages back to the generator. This initiates an optimization process, creating a dynamic interaction loop between the generator and executor. This loop continues until a successful build solution is achieved or the maximum number of iterations is reached. Additionally, the executor implements an LLM-based discriminator on the build instructions and logs. This ensures the success of the build and helps identify and resolve errors comprehensively.

A critical challenge in designing the executor module is accurately verifying whether the project has been successfully built. We employ the Python Docker SDK to capture the execution results within the Docker container and save these as log files. However, certain build instruction errors may lead to issues that Docker cannot detect. One such scenario arises when a project implements custom error handling, which may suppress the generation of error messages. For example, in *LocalAI*, the *Makefile* includes error handling for build targets, meaning that even if the wrong target is selected, Docker will not report any build errors. Another issue occurs when the Dockerfile generated by the LLM lacks essential build instructions (e.g., '*make*'). In this case, while no errors

may be reported, no actual building operation takes place. We refer to these situations as "non-error failures". Due to the diverse nature of the outputs in these cases, traditional rule-based or keyword-matching error detection methods often fail to reliably identify such build failures.

To address the challenge, we design an LLM-based discriminator to identify these build failures. In designing the LLM discriminator, we incorporate two key insights from our manual construction process: Static criterion: The Dockerfile should include build and compile instructions (e.g., '*make*', '*cmake –build*'), and the build target must match the default or primary components as described in the project's documentation. Dynamic criterion: We store log files (an example is available in our project [56]) generated during the build process. By analyzing these logs, we can confirm whether the build commands are executed successfully. Logs from successful builds typically show compile progress (e.g., '*[3%] Building CXX object..*') and test progress (e.g., '*Performing Test C_FLAG_WALL Success*'). The discriminator's judgment process is divided into two steps. First, we design prompts to guide the LLM in making judgments based on these two key criteria. Second, to further mitigate hallucinations, we introduce a reflection mechanism to re-validate the "judgment process" of the first step. If the "judgment process" did not strictly adhere to the two criteria, the build is deemed a failure, thus minimizing FP. When providing information to the discriminator, the executor carefully controls the context length and selects the log segments most relevant to state determination. In the case of "error-type failures", which are typically direct and concise, the executor inputs the most recent 50 execution logs into the LLM for accurate error detection and analysis. When no errors are reported, and since determining "non-error failures" often requires more contextual information, the executor inputs the Dockerfile and the last 200 lines of the log. If the input exceeds the LLM's context length limit, a sliding window mechanism is used, prioritizing the retention of the most recent logs to ensure effective resolution of new errors.

To evaluate the effectiveness of the LLM-based discriminator, we examine the accuracy of 4 LLMs—*DeepSeek-v2*, *DeepSeek-v3*, *GPT-4o*, and *GPT-4o mini*—on the Top100 dataset. We manually check and validate the discriminator's judgments during the build process. Out of the 400 build processes, 249 are classified as successful. We manually verify these 249 samples and find all judgments to be correct. Regardless of the LLM used, the discriminator accurately identifies all successful builds. This validates the effectiveness of the LLM-based discriminator design.

4.5 Prompt Design in `CXXCrafter`

We have designed a set of prompts tailored to 5 specific scenarios, including build system and entry point identification, RAG for documentation parsing, initial Dockerfile creation, Dockerfile modification, and build success discriminator. These prompts, developed based on expert knowledge and refined through iterative experimentation, incorporate strategies such as nested prompt templates and RAG to address task complexity. The complete set of prompts is provided in the appendix file[56]. In the design process, several challenges arise when prompting LLMs to effectively complete building tasks. **Challenge 1** involves breaking down complex problems when generating build solutions. We address this by using embedded prompt templates to dynamically inject parsing information, such as in our Dockerfile generation prompt (Section 2.3 in Appendix file [56]), which dynamically fills in parsed data. Additionally, we provide the LLM with strategic guidance in the form of requirement notes. **Challenge 2** stems from unclear project-specific build processes and details. To resolve this, we utilize RAG to retrieve relevant files from the project's source code directory, such as documentation RAG and build system identification prompts (Section 2.1 and 2.2 in Appendix file[56]). Finally, **Challenge 3**, related to token limitations, arises during Dockerfile modification. To effectively manage error feedback, we retain error messages and decisions within a single session to ensure continuity. However, when the context exceeds the token limit, we remove the earliest resolved issues to maintain focus on the current task.

An Example of Prompt. As shown in Figure 4, this prompt is used to generate a Dockerfile. Specifically, we combine information obtained from the parser with pre-defined templates for built-in prompts to create the final prompt that generates the Dockerfile. The prompts corresponding to numbers 3, 4, and 5 in the figure include the information parsed by the parser. Prompt 1 is a Dockerfile template that guides the LLM to structure the Dockerfile correctly, breaking down steps such as basic environment setup and dependency installation. In addition, this prompt includes specific requirements, such as correctly handling line breaks in comments, to ensure that the generated Dockerfile is free from syntax errors.

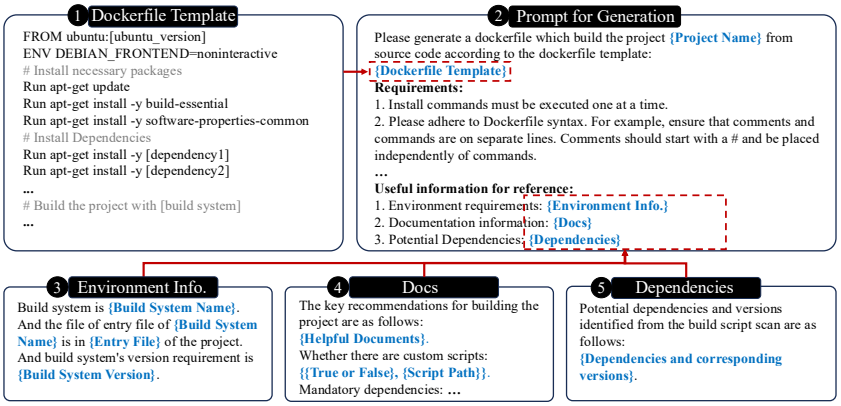

Fig. 4. Prompt of the Generator Module. The prompts corresponding to numbers 3, 4, and 5 are from the Parser's results, with blue text indicating dynamically parsed content.

5 Evaluation

We implement `CXXCrafter` in Python, without relying on LLM frameworks like *Langchain*. Our implementation ensures a clear modular structure and strong scalability, enabling easy upgrades and replacements of components and tools. `CXXCrafter` consists of 1,664 lines of code and uses 5 different types of prompts. In the experiments, `CXXCrafter` uses *GPT-4o* as the default LLM, with the dynamic interaction limit set to 10 by default. The execution environment for our build solution is managed through Python Docker SDK. Our experiments are conducted on three *Ubuntu 22.04* servers with varying hardware configurations. The first machine is equipped with two Intel Xeon 6330 processors, 512 GB of RAM, and 3 TB of HDD storage. The second and third machines feature four Intel Xeon 8260 processors, 256 GB of RAM, and 3.37 TB of HDD storage each.

Research Questions. Our evaluation aims to address the following research questions:

- **RQ3 (Effectiveness)**: How many C/C++ projects can be automatically built by `CXXCrafter`?
- **RQ4 (Ablation Study)**: How does each component within `CXXCrafter` contribute to the overall build performance?
- **RQ5 (Case Study)**: How does `CXXCrafter` resolve build issues that manual methods fail to address, and what specific advantages does it offer in handling complex C/C++ projects?
- **RQ6 (Efficiency and Cost)**: What is the efficiency and cost of using CXXCrafter?

Dataset. Two datasets are used for evaluation. The first dataset, Top100, is described in Section 3. The second dataset, from Awesome-CPP [7], includes a broader collection with 58.6K stars as of September 2024. It covers a wide range of C++ libraries, frameworks, and tools, providing a comprehensive testbed for evaluating `CXXCrafter` 's performance across diverse real-world C/C++

projects. To ensure there are no duplicates between datasets and that all projects are buildable, we remove any overlapping projects with the Top100 dataset and manually exclude non-C/C++ projects based on the criteria outlined in Section 3. After filtering, 652 distinct projects remain for evaluation. For all projects, we use the latest available version for experimentation.

LLMs Selection. We select four LLMs: *GPT-4o*, a high-performance closed-source model; *GPT-4o mini*, a more affordable alternative of *GPT-4o*; *DeepSeek-v2* with 236B parameters and *DeepSeek-v3* with 671B parameters, both open-source models that excel in code-related tasks.

Baselines. We select 3 types of baselines. (1) **Default Build Commands**: We have collected over 20 common C/C++ build systems and their associated instructions (see Appendix [56]). Based on this collection, we develop an automated script to execute default or commonly used build commands. The script first identifies potential configuration files, such as *Makefile* or *CMakeLists.txt*. It then identifies all possible build systems from the configuration files and executes their corresponding build instructions in sequence. (2) **Programmers**: The manual building methods used in Section 3.1. (3) **Different Bare LLMs**: We also explore the performance of different bare LLMs. These models use the same prompts as the `CXXCrafter` generator but lack the information provided by `CXXCrafter`'s parser. Additionally, there is no dynamic iterative process if the build fails.

Metrics of Success. We determine the success of builds by manually inspecting the Dockerfile instructions and the corresponding execution outputs. During this inspection, we follow two criteria to efficiently assess success as follows: (1) Static Criterion: The Dockerfile must contain the necessary build-related instructions, and the build target should align with the primary components as specified in the project documentation. (2) Dynamic Criterion: We analyze the execution logs generated during the building process to ensure that the build commands are executed properly and that the process completes without errors. Only projects that satisfy both criteria are considered as successful builds. We further evaluate these metrics (see Section 6), confirming that builds meeting these criteria yield outputs consistent with those produced by manual builds and demonstrate correct functionality. These criteria are the same as those in the executor (see Section 4.4), with the key difference being that we perform manual checks to prevent misjudgments by LLMs.

5.1 Overall Effectiveness Evaluation (RQ3)

To address **RQ3**, experiments are conducted on two datasets. For `CXXCrafter`, the default dynamic interaction step limit is set to 10, with *GPT-4o* serving as the core LLM due to its superior performance in trials. We also evaluate the build performance of `CXXCrafter` using another powerful open-source model, *DeepSeek-v3*, while keeping all other settings the same. Additionally, we compare the results with those of the Default Build Commands and the bare LLMs, as mentioned above. For all build results, we manually inspect and verify their correctness.

As shown in Table 5, `CXXCrafter` demonstrates significant superiority. For the Top100 dataset, `CXXCrafter (Default)` successfully builds 75 projects, significantly surpassing other methods. The Default Build Commands tool achieves 21 builds, while the bare LLM models, show a similar performance, with 23 and 17 successful builds, respectively. In the Awesome-CPP collection, `CXXCrafter` achieves 512 successful builds, greatly outperforming Default Build Commands (272 builds) and the bare LLMs (264 for *DeepSeek-v3* and 215 for *GPT-4o*).

The Default Build Commands approach achieves a 39.01% success rate. While this method proves reliable for simpler projects, it struggles with more complex or non-standard build configurations, resulting in a relatively low success rate. The bare LLMs (*DeepSeek*, *GPT-4o*, and *GPT-4o mini*) demonstrate even lower success rates of 38.43%, 31.65%, and 19.81%, respectively. These findings suggest that while LLMs have some capacity to handle build tasks, their effectiveness remains limited without further domain-specific optimization. In some cases, they perform worse than

rule-based methodologies. Notably, *GPT-4o mini*, with a 19.81% success rate, exhibits significant limitations when applying a smaller LLM to complex build processes.

In stark contrast, CXXCrafter achieves a 78.10% success rate, showing a marked improvement over all other methods. This outcome underscores the effectiveness of CXXCrafter 's modular design, which allows it to adapt efficiently to diverse build scenarios. The substantial gap between CXXCrafter and the other methodologies emphasizes the importance of specialized agent in automating complex tasks like C/C++ project builds.

Overall, CXXCrafter significantly outperforms both the bare LLMs and the heuristic build tool, demonstrating high success rates and the potential to reduce the time and efforts required for large-scale OSS building, making it a valuable tool in modern development workflows.

Finding 4: Without a carefully designed iterative framework, LLMs remain inadequate for addressing the inherent complexity and multi-stage processes of project building.

Table 5. Experimental Results Between CXXCrafter and Baselines.

Methodology	Top100 Builds	Awesome-CPP Builds	Total Builds	Success Rate (%)
Programmers	86	N/A	N/A	N/A
Default Build Commands	21	272	293	39.01
Bare LLM (*DeepSeek-v2*)	23	239	262	34.22
Bare LLM (*DeepSeek-v3*)	25	264	289	38.43
Bare LLM (*GPT-4o mini*)	17	132	149	19.81
Bare LLM (*GPT-4o*)	23	215	238	31.65
CXXCrafter (*DeepSeek-v3*)	67	510	577	76.73
CXXCrafter (Default)	**75**	**512**	**587**	**78.10**

5.2 Component Design Analysis (RQ4)

In this section, we present a detailed component-wise analysis to assess the contribution of key modules and various configurations in CXXCrafter. This analysis focuses on 3 main aspects:

- The role of the **parser** module in enhancing build success.
- The impact of **dynamic interaction** and effect of varying dynamic interaction step counts.
- The impact of **different LLMs** on CXXCrafter 's performance.

We conduct experiments on the Top100 dataset, with results shown in Figure 5. The CXXCrafter (Default) also uses *GPT-4o* as the LLM, with a maximum of 10 dynamic interaction steps.

The Role of the Parser. The default configuration, with all components enabled, achieves the highest number of successful builds, completing 75 builds. When the parser is removed (CXXCrafter-w/o-Parser), the success rate drops to 48 builds, highlighting the parser's crucial role. In CXXCrafter, build system selection and entry file identification rely on the parser, which forms the foundation for the entire build process and helps avoid many errors. Additionally, build-related documentation is crucial. The parser automates the search for and interpretation of these documents, further enhancing the build success rate.

The Impact of Dynamic Interaction. Dynamic interaction is the key design of CXXCrafter, allowing iterative execution and modification during the build process. When dynamic interaction is disabled (CXXCrafter-w/o-Interaction), the number of successful builds drops sharply to 22, highlighting its importance in managing complex, multi-step build scenarios. We also analyze the impact of different interaction step limits. When the limit is set to 5 steps, performance declines, with only 69 successful builds. Increasing the step count to 20 does not further improve performance, with 74 successful builds. We observe that the benefits of increasing interaction steps begin to diminish beyond a certain threshold. For example, increasing the step count from 0 to 5 leads to a

significant improvement of 47 successful builds. However, increasing it from 5 to 10 only adds 6 builds. Furthermore, increasing from 10 to 20 results in one fewer successful build. This variation is likely caused by the inherent instability in the LLM's output.

Finding 5: Dynamic interaction plays a crucial role in managing multi-step tasks in the agent design. Increasing interaction steps improves success rates while the enhancement can be limited.

The Impact of Different LLMs. Finally, we evaluate the impact of different LLMs on `CXXCrafter`. Specifically, *DeepSeek-v2* completes 57 builds, *DeepSeek-v3* completes 67, while *GPT-4o mini* completes 50. *GPT-4o* remains the most effective, with 75 successful builds. These results highlight the significant impact of LLMs on `CXXCrafter`'s ability to automate the build process. Notably, we observe that open-source LLMs can now achieve performance on par with leading closed-source models. Furthermore, cost-effective closed-source models like *GPT-4o mini* can achieve about 50% of the effectiveness in our design. Additionally, `CXXCrafter`, based on these models, performs much better as an agent than bare LLMs (see Section 5.1), further demonstrating that our design leads to a substantial improvement in performance.

Finding 6: The selection of LLMs significantly affects the agent's performance. More powerful models, such as *GPT-4o*, can offer stronger assistance and enhance the overall effectiveness.

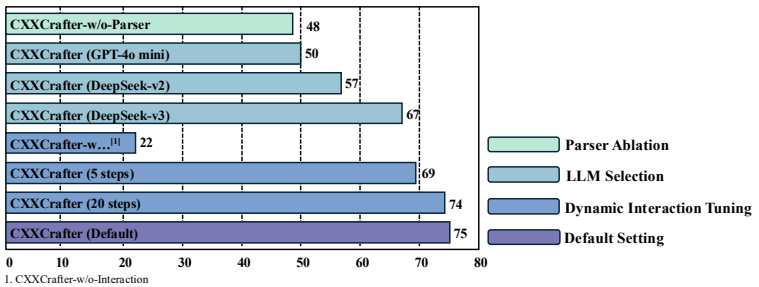

Fig. 5. Number of Successful Builds of `CXXCrafter` Variants on the Top100 Dataset.

5.3 Case Study (RQ5)

Among the Top100 dataset, 72 projects are successfully built manually as well as by `CXXCrafter` in Section 5.1. Three projects succeed in `CXXCrafter` but fail in manual builds, while 14 projects are successful manually but failed by `CXXCrafter`.

We analyze the 3 cases where manual builds failed. **Case 1**: When building *CQuery* manually, an error occurs with "*std::unique_lock* being defined in the header *<mutex>*", suggesting that "*<mutex>*" is not included. Initially, we suspect this is related to the *Clang* version. However, after trying various versions (e.g., *Clang* 7, *Clang* 11), the issue remains. Further analysis reveals that the *CQuery* build script defaults to *Clang* 7 on Ubuntu 14.04, which is incompatible with Ubuntu 22.04. `CXXCrafter` suggests using Ubuntu 20.04, where *Clang* 7 is compatible and thus resolve the issue. **Case 2**: In the *Paddle* project, human attempts produce the error: "*libdnnl.so*: undefined reference to *dnnl::impl...*". Despite confirming *oneDNN* installations and testing various versions, the error persists. `CXXCrafter` identifies a version mismatch between *protobuf* and *oneDNN* (requiring *protobuf* 3.20.2+ as noted in the "requirements.txt"), a detail overlooked by human builders. **Case 3**: In *DOOM*, a linking error initially suggests issues with 32-bit libraries or the container environment,

leading to various adjustments. CXXCrafter identifies the actual issue as a mismatch between the *TSL* variable *errno* and the shared library version, resolving it with a code modification.

CXXCrafter's Advantages over Humans. CXXCrafter offers two key advantages over manual building: (i) The parser module uses RAG to efficiently process documents and other information, allowing it to identify build-related information more comprehensively than manual searches. For example, in CASE2, CXXCrafter prevented errors that would have arisen from overlooking crucial information during manual builds. (ii) The LLM stores historical build knowledge, compensating for the limitations of human experience. As demonstrated in CASE3, CXXCrafter makes more correct decisions, avoiding potential errors. The major drawback of CXXCrafter is its higher error rate when installing complex dependencies, such as 'CUDA' for *OpenPose*. These libraries involve complex installation processes with many dependencies and steps. This may be resolved in the future through knowledge injection or RAG.

5.4 Efficiency and Cost (RQ6)

We assess the cost of CXXCrafter across three dimensions: time, financial expense, and disk storage. These factors are crucial in determining the practical usefulness and scalability of our approach.

Time Cost. On average, CXXCrafter takes 875.31 seconds to successfully build a project on the Top100 dataset. Additionally, the average time cost for the failed projects is 2.67 hours. However, time costs can vary due to factors such as Docker caching and network speed. Enabling multi-processing significantly enhances efficiency, substantially reducing the overall build time.

Financial Cost. Running CXXCrafter on the Top100 dataset generates 4,297,652 input tokens and 624,170 output tokens. This incurs *GPT-4o* related costs of $21.49 for input tokens and $9.36 for output tokens, respectively. Among these, 75 projects are successfully built, with an average cost per successful build calculated at $0.41. The 25 failed projects generate a total of 2,420,092 input tokens and 225,154 output tokens, with an average cost of $0.6191 per project. This price is based on OpenAI's pricing as of September 2024.

Disk Storage Cost. The experiment generates over 50 TB of data, including Docker container caches and image files. This creates significant storage demands. Despite using three machines, disk space management remains a critical and recurring challenge throughout the experiment.

6 Discussion

Effectiveness and Consistency of Build Artifacts. We conduct an in-depth analysis of the build artifacts to verify their functionality and consistency with manually built artifact. To verify that the build artifacts perform as expected, we run the unit tests provided by the projects. Among the 75 successfully built projects in TOP100, we identify 24 that generate test executables. Among them 22 projects pass while 2 fail due to missing audio connections in *libsoundio* and lack of GUI display support for *Stockfish* on our server (due to the absence of the relevant devices). These results confirm that the build artifacts produced by CXXCrafter are valid and function as intended. Additionally, we use a diff tool to compare the automated build artifacts with the manually built artifacts for all 75 successfully built projects. The results show that the automated and manual build artifacts are completely consistent. Detailed experimental results can be found in our Project [56]. These results also further validate the effectiveness of our success metrics in the experiment.

Building Different Software Versions. We conduct two additional experiments. First, we investigate the build success rate across different software versions. For 20 projects that are successfully built, we randomly select 5 commits for each, covering their entire repository commit histories from the creation of the repository. CXXCrafter achieves an 81% success rate with 81 out of 100 builds successful. This demonstrates that CXXCrafter are effective across multiple versions. Some failures occur due to older versions requiring outdated packages, which are often hard to

find. Second, we test the build performance of consecutive commits (i.e., building one commit after another). By selecting the latest 5 commits from 20 projects, we observe a higher success rate, with 96 out of 100 builds successful. Overall, these experiments demonstrate that CXXCrafter is effective in both version diversity and consecutive commit builds.

Building Different Language Projects. CXXCrafter's design shows promising potential for other languages. Specifically, we conduct a simple migration to the top 100 most starred Java projects (after filtering the unbuildable projects, 76 remain), successfully building 57 out of 76 projects, achieving a success rate of 75%. This already represents promising performance in Java automation build methods [16] to our best knowledge.

Potential Applications of CXXCrafter. CXXCrafter is highly beneficial for various downstream applications in software security analysis, including but not limited to: (1) reproducing identified vulnerabilities by facilitating set up environments with specific versions for vulnerability reproduction; (2) static program analysis, particularly high-precision analysis based on *LLVM IR*, which often requires code to meet compilation requirements. CXXCrafter can assist in fulfilling this process; (3) dynamic program analysis, such as source code instrumentation, where CXXCrafter can ensure proper compilation, thus streamlining workflows for tasks like fuzz testing.

7 Threats to Validity

Our study mainly suffers from the following threats to validity. Specifically, the internal validity threat in our study mainly stems from the variations in LLM performance, which could impact the experimental results. To mitigate this issue, we conduct experiments using the open-source model *DeepSeek*. Additionally, during the dependency download process, CXXCrafter retrieves dependencies from online sources (e.g., by using '*apt*' to install packages or '*git*' to download repositories). If these sources become unavailable or if network issues arise, it may affect the results. Furthermore, updates to the software itself could also introduce internal validity threats. Our research primarily targets popular projects, for which LLMs may have gain deeper understanding and the documentation is usually more comprehensive. Therefore, CXXCrafter's performance on less popular projects may be impacted, which constitutes an external validity threat. To address this, we plan to further incorporate RAG techniques or retrain the model in the future. Lastly, the authors have rich experiences in C/C++ related researches, and via further extensive investigations in C/C++ projects build automation, the authors have gained deep understandings towards build-related issues. As a result, we believe this study' threats of construct validity are limited.

8 Conclusion

This paper presents CXXCrafter, an LLM-based agent system that automates the C/C++ build process, marking the first exploration of using LLM agents for this task. Through an empirical study of popular open-source C/C++ projects, we identify and categorize 384 build errors, providing insights into the key challenges of automating C/C++ builds. CXXCrafter addresses these challenges by dynamically managing dependencies, resolving build issues, and diagnosing errors, achieving a success rate of 78% across 752 projects. Our system advances build automation and offers significant support for downstream program analysis tasks, such as vulnerability research and performance optimization. Future work focuses on refining and expanding CXXCrafter's capabilities to handle more complex build scenarios and support downstream program analysis tasks.

9 Data Availability

We release the implementation and all associated publicly available data in the website [56].

Acknowledgment

We would like to thank the anonymous reviewers for their insightful comments that helped improve the quality of the paper. This work was supported in part by the National Natural Science Foundation of China (U2436207, 62172105, 62372193). Yuan Zhang and Min Yang are the corresponding authors. Yuan Zhang was supported in part by the Shanghai Pilot Program for Basic Research - FuDan University 21TQ1400100 (21TQ012). Min Yang is a faculty of Shanghai Institute of Intelligent Electronics & Systems, and Engineering Research Center of Cyber Security Auditing and Monitoring, Ministry of Education, China.

References

[1] Apache Ant Project. 2024. Apache Ant. https://ant.apache.org/. [Online; accessed 29-Aug-2024].
[2] Apache Maven Project. 2024. Apache Maven. https://maven.apache.org/. [Online; accessed 10-Sep-2024].
[3] Saikat Barua. 2024. Exploring autonomous agents through the lens of large language models: A review. *arXiv preprint arXiv:2404.04442* (2024).
[4] Nicholas Carlini, Florian Tramer, Eric Wallace, Matthew Jagielski, Ariel Herbert-Voss, Katherine Lee, Adam Roberts, Tom Brown, Dawn Song, Ulfar Erlingsson, et al. 2021. Extracting training data from large language models. In *30th USENIX Security Symposium (USENIX Security 21)*. 2633–2650.
[5] 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)*. 847–864.
[6] Zakir Durumeric, Frank Li, James Kasten, Johanna Amann, Jethro Beekman, Mathias Payer, Nicolas Weaver, David Adrian, Vern Paxson, Michael Bailey, and J. Alex Halderman. 2014. The Matter of Heartbleed. In *Proceedings of the 2014 Conference on Internet Measurement Conference* (Vancouver, BC, Canada) *(IMC '14)*. Association for Computing Machinery, New York, NY, USA, 475–488. https://doi.org/10.1145/2663716.2663755
[7] Faraz Fallahi. 2024. Awesome C++. https://github.com/fffaraz/awesome-cpp. Accessed: 2024-09-11.
[8] Chongzhou Fang, Ning Miao, Shaurya Srivastav, Jialin Liu, Ruoyu Zhang, Ruijie Fang, Asmita, Ryan Tsang, Najmeh Nazari, Han Wang, and Houman Homayoun. 2024. Large Language Models for Code Analysis: Do LLMs Really Do Their Job? arXiv:2310.12357 [cs.SE] https://arxiv.org/abs/2310.12357
[9] Free Software Foundation. 2024. *GNU Compiler Collection.* Free Software Foundation. https://gcc.gnu.org/ Accessed: 1-Aug-2024.
[10] Free Software Foundation. 2023. *GNU Make: The GNU Make Manual.* Free Software Foundation. https://www.gnu.org/software/make/manual/make.html Accessed: 26-Feb-2023.
[11] Free Software Foundation. 2024. *Autotools Introduction.* Free Software Foundation. https://www.gnu.org/software/automake/manual/html_node/Autotools-Introduction.html Accessed: 12-Sep-2024.
[12] GitHub. 2021. GitHub Copilot: Your AI pair programmer. https://github.com/features/copilot Accessed: 2024-09-13.
[13] Gradle, Inc. 2024. Gradle Build Tool. https://gradle.org/. [Online; accessed 9-Sep-2024].
[14] Konstantin Grotov, Sergey Titov, Yaroslav Zharov, and Timofey Bryksin. 2024. Untangling Knots: Leveraging LLM for Error Resolution in Computational Notebooks. arXiv:2405.01559 [cs.SE] https://arxiv.org/abs/2405.01559
[15] Lukas Gygi. 2021. *CppBuild: Large-Scale, Automatic Build System for Open Source C++ Repositories.* B.S. thesis. ETH Zurich.
[16] Foyzul Hassan, Shaikh Mostafa, Edmund SL Lam, and Xiaoyin Wang. 2017. Automatic building of java projects in software repositories: A study on feasibility and challenges. In *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. IEEE, 38–47.
[17] Foyzul Hassan, Rodney Rodriguez, and Xiaoyin Wang. 2018. RUDSEA: recommending updates of Dockerfiles via software environment analysis. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering* (Montpellier, France) *(ASE '18)*. Association for Computing Machinery, New York, NY, USA, 796–801. https://doi.org/10.1145/3238147.3240470
[18] Foyzul Hassan and Xiaoyin Wang. 2018. HireBuild: An Automatic Approach to History-Driven Repair of Build Scripts. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. 1078–1089. https://doi.org/10.1145/3180155.3180181
[19] Glen Hopkins and Kristjan Kalm. 2023. Classifying complex documents: comparing bespoke solutions to large language models. arXiv:2312.07182 [cs.CL] https://arxiv.org/abs/2312.07182
[20] Eric Horton and Chris Parnin. 2019. Dockerizeme: Automatic inference of environment dependencies for python code snippets. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 328–338.

[21] Lars Huning and Elke Pulvermueller. 2021. Automatic Code Generation of Safety Mechanisms in Model-Driven Development. *Electronics* 10, 24 (2021). https://doi.org/10.3390/electronics10243150

[22] Kitware Inc. 2024. *CMake: A Cross-Platform Makefile Generator*. Kitware Inc. https://cmake.org/ Accessed: 12-Sep-2024.

[23] Inc. JFrog. 2024. Conan: C and C++ Open Source Package Manager. https://conan.io/. Accessed: 11-Sep-2024.

[24] Xin Jiang, Xiang Li, Wenjia Ma, Xuezhi Fang, Yiqun Yao, Naitong Yu, Xuying Meng, Peng Han, Jing Li, Aixin Sun, and Yequan Wang. 2024. Sketch: A Toolkit for Streamlining LLM Operations. arXiv:2409.03346 [cs.CL] https://arxiv.org/abs/2409.03346

[25] Matthew Jin, Syed Shahriar, Michele Tufano, Xin Shi, Shuai Lu, Neel Sundaresan, and Alexey Svyatkovskiy. 2023. Inferfix: End-to-end program repair with llms. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1646–1656.

[26] Jean Kaddour, Joshua Harris, Maximilian Mozes, Herbie Bradley, Roberta Raileanu, and Robert McHardy. 2023. Challenges and applications of large language models. *arXiv preprint arXiv:2307.10169* (2023).

[27] Cheryl Lee, Chunqiu Steven Xia, Jen-tse Huang, Zhouruixin Zhu, Lingming Zhang, and Michael R Lyu. 2024. A Unified Debugging Approach via LLM-Based Multi-Agent Synergy. *arXiv preprint arXiv:2404.17153* (2024).

[28] Xiao Liu, Hao Yu, Hanchen Zhang, Yifan Xu, Xuanyu Lei, Hanyu Lai, Yu Gu, Hangliang Ding, Kaiwen Men, Kejuan Yang, et al. 2023. Agentbench: Evaluating llms as agents. *arXiv preprint arXiv:2308.03688* (2023).

[29] LLVM Project. 2024. The LLVM Compiler Infrastructure Project. https://llvm.org/. [Online; accessed 18-Jun-2024].

[30] Yiling Lou, Zhenpeng Chen, Yanbin Cao, Dan Hao, and Lu Zhang. 2020. Understanding build issue resolution in practice: symptoms and fix patterns. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Virtual Event, USA) *(ESEC/FSE 2020)*. Association for Computing Machinery, New York, NY, USA, 617–628. https://doi.org/10.1145/3368089.3409760

[31] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, Ming Gong, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie Liu. 2021. CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation. arXiv:2102.04664 [cs.SE] https://arxiv.org/abs/2102.04664

[32] Ruchika Malhotra, Anjali Bansal, and Marouane Kessentini. 2024. A Systematic Literature Review on Maintenance of Software Containers. *ACM Comput. Surv.* 56, 8, Article 193 (apr 2024), 38 pages. https://doi.org/10.1145/3645092

[33] Inc. Microsoft. 2024. vcpkg: C++ Library Manager for Windows, Linux, and macOS. https://vcpkg.io/. Accessed: 23-Aug-2024.

[34] Manisha Mukherjee and Vincent J. Hellendoorn. 2024. "Medium" LMs of Code in the Era of LLMs: Lessons From StackOverflow. arXiv:2306.03268 [cs.CL] https://arxiv.org/abs/2306.03268

[35] Daniel Munoz. 2015. *Why the C programming language still rules.* Retrieved July 21, 2015 from https://www.toptal.com/c/after-all-these-years-the-world-is-still-powered-by-c-programming

[36] Yoonjong Na, Seunghoon Woo, Joomyeong Lee, and Heejo Lee. 2024. CNEPS: A Precise Approach for Examining Dependencies among Third-Party C/C++ Open-Source Components. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering* (Lisbon, Portugal) *(ICSE '24)*. Association for Computing Machinery, New York, NY, USA, Article 236, 12 pages. https://doi.org/10.1145/3597503.3639209

[37] Olivier Nourry, Yutaro Kashiwa, Weiyi Shang, Honglin Shu, and Yasutaka Kamei. 2024. My Fuzzers Won't Build: An Empirical Study of Fuzzing Build Failures. *ACM Trans. Softw. Eng. Methodol.* (aug 2024). https://doi.org/10.1145/3688842 Just Accepted.

[38] npm, Inc. 2023. npm Documentation. https://docs.npmjs.com/. [Online; accessed 23-Oct-2023].

[39] LLVM Project. 2024. *Clang: A C Language Family Frontend for LLVM.* LLVM Foundation. https://clang.llvm.org/ Accessed: 19-Jun-2024.

[40] Python Packaging Authority (PyPA). [n. d.]. setuptools: Easily Download, Build, Install, Upgrade, and Uninstall Python Packages. https://setuptools.pypa.io/. Accessed: 2024-09-12.

[41] Rajat Rawat, Hudson McBride, Dhiyaan Nirmal, Rajarshi Ghosh, Jong Moon, Dhruv Alamuri, Sean O'Brien, and Kevin Zhu. 2024. DiversityMedQA: Assessing Demographic Biases in Medical Diagnosis using Large Language Models. arXiv:2409.01497 [cs.CL] https://arxiv.org/abs/2409.01497

[42] IBM Research. 2023. LLM-based AI agents are what's next. https://research.ibm.com/blog/what-are-ai-agents-llm Accessed: 2024-09-13.

[43] Giovanni Rosa, Antonio Mastropaolo, Simone Scalabrino, Gabriele Bavota, and Rocco Oliveto. 2023. Automatically Generating Dockerfiles via Deep Learning: Challenges and Promises. arXiv:2303.15990 [cs.SE] https://arxiv.org/abs/2303.15990

[44] Alon Talmor, Jonathan Herzig, Nicholas Lourie, and Jonathan Berant. 2019. CommonsenseQA: A Question Answering Challenge Targeting Commonsense Knowledge. arXiv:1811.00937 [cs.CL] https://arxiv.org/abs/1811.00937

[45] Wei Tang, Zhengzi Xu, Chengwei Liu, Jiahui Wu, Shouguo Yang, Yi Li, Ping Luo, and Yang Liu. 2022. Towards understanding third-party library dependency in c/c++ ecosystem. In *Proceedings of the 37th IEEE/ACM International*

Conference on Automated Software Engineering. 1–12.

[46] Xiaojuan Tang, Zilong Zheng, Jiaqi Li, Fanxu Meng, Song-Chun Zhu, Yitao Liang, and Muhan Zhang. 2023. Large Language Models are In-Context Semantic Reasoners rather than Symbolic Reasoners. arXiv:2305.14825 [cs.CL] https://arxiv.org/abs/2305.14825

[47] MSRC Team. 2019. *We need a safer systems programming language.* Retrieved July 18, 2019 from https://msrc.microsoft.com/blog/2019/07/we-need-a-safer-systems-programming-language/

[48] The pip developers. 2024. https://pypi.org/project/pip/. Version 24.2; Accessed: 29-Jul-2024.

[49] Gladys Tyen, Hassan Mansoor, Victor Cărbune, Peter Chen, and Tony Mak. 2024. LLMs cannot find reasoning errors, but can correct them given the error location. arXiv:2311.08516 [cs.AI] https://arxiv.org/abs/2311.08516

[50] Eric Wallace, Kai Xiao, Reimar Leike, Lilian Weng, Johannes Heidecke, and Alex Beutel. 2024. The Instruction Hierarchy: Training LLMs to Prioritize Privileged Instructions. arXiv:2404.13208 [cs.CR] https://arxiv.org/abs/2404.13208

[51] Xingyao Wang. 2024. OpenDevin: An Open Platform for AI Software Developers as Generalist Agents. https://github.com/OpenDevin/OpenDevin Accessed: 2024-09-13.

[52] Mats Wichmann. 2024. *SCons: Software Construction Tool.* https://scons.org/ Accessed: 12-Sep-2024.

[53] Wikipedia contributors. 2024. Software build — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/wiki/Software_build. [Online; accessed 11-Sep-2024].

[54] Seunghoon Woo, Sunghan Park, Seulbae Kim, Heejo Lee, and Hakjoo Oh. 2021. CENTRIS: A Precise and Scalable Approach for Identifying Modified Open-Source Software Reuse. arXiv:2102.06182 [cs.SE] https://arxiv.org/abs/2102.06182

[55] Jiahui Wu, Zhengzi Xu, Wei Tang, Lyuye Zhang, Yueming Wu, Chengyue Liu, Kairan Sun, Lida Zhao, and Yang Liu. 2023. OSSFP: Precise and Scalable C/C++ Third-Party Library Detection using Fingerprinting Functions. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. 270–282. https://doi.org/10.1109/ICSE48619.2023.00034

[56] Yuremin. 2025. Yuremin/CXXCrafter-Community-Edition: CXXCrafter_v1.0.0-alpha (v1.0.0). https://doi.org/10.5281/zenodo.15273210

[57] Bo Zhang, Hui Ma, Jian Ding, Jian Wang, Bo Xu, and Hongfei Lin. 2024. Distilling Implicit Multimodal Knowledge into LLMs for Zero-Resource Dialogue Generation. arXiv:2405.10121 [cs.CL] https://arxiv.org/abs/2405.10121

[58] Chen Zhang, Bihuan Chen, Linlin Chen, Xin Peng, and Wenyun Zhao. 2019. A large-scale empirical study of compiler errors in continuous integration. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Tallinn, Estonia) *(ESEC/FSE 2019)*. Association for Computing Machinery, New York, NY, USA, 176–187. https://doi.org/10.1145/3338906.3338917

[59] Chen Zhang, Bihuan Chen, Junhao Hu, Xin Peng, and Wenyun Zhao. 2023. BuildSonic: Detecting and Repairing Performance-Related Configuration Smells for Continuous Integration Builds. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering* (Rochester, MI, USA) *(ASE '22)*. Association for Computing Machinery, New York, NY, USA, Article 18, 13 pages. https://doi.org/10.1145/3551349.3556923

[60] Chen Zhang, Bihuan Chen, Xin Peng, and Wenyun Zhao. 2022. BuildSheriff: change-aware test failure triage for continuous integration builds. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 312–324. https://doi.org/10.1145/3510003.3510132

[61] Zibin Zheng, Kaiwen Ning, Yanlin Wang, Jingwen Zhang, Dewu Zheng, Mingxi Ye, and Jiachi Chen. 2024. A Survey of Large Language Models for Code: Evolution, Benchmarking, and Future Trends. arXiv:2311.10372 [cs.SE] https://arxiv.org/abs/2311.10372

[62] Anni Zou, Wenhao Yu, Hongming Zhang, Kaixin Ma, Deng Cai, Zhuosheng Zhang, Hai Zhao, and Dong Yu. 2024. DOCBENCH: A Benchmark for Evaluating LLM-based Document Reading Systems. arXiv:2407.10701 [cs.CL] https://arxiv.org/abs/2407.10701

Received 2025-02-26; accepted 2025-04-01