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

This appendix provides additional details and supporting information referenced in the main body of the paper.

1 BUILD SYSTEM LIST

Table 1. Build Systems and Their Default Commands

Build System	**Description**	**Default Command**
Autotools	A suite of programming tools designed to assist in making source code packages portable to many Unix-like systems.	./configure && make
Bazel	A multi-language, fast and scalable build system from Google.	bazel build
Bear	A tool to generate compilation database for clang tooling.	bear –
Buck	A fast build system that encourages the creation of small, reusable modules over a variety of platforms and languages including C++ developed and used at Facebook. Written in Java.	buck build <path_to_target>
build2	An open source, cross-platform build toolchain that aims to approximate Rust Cargo’s convenience for developing and packaging C/C++ projects while providing more depth and flexibility, especially in the build system.	b
ccache	A fast C/C++ compiler cache.	\
clib	Package manager for the C programming language.	clib install <package_name>
CMake	Cross-platform free and open-source software for managing the build process of software using a compiler-independent method.	mkdir build && cd build && cmake .. && make
Cget	CMake package retrieval.	cget install <package_name>
Conan	C/C++ Package Manager, open-sourced.	conan install .
CPM	A C++ Package Manager based on CMake and Git.	\
FASTBuild	High performance, open-source build system supporting highly scalable compilation, caching and network distribution.	fbuild
Hunter	CMake driven cross-platform package manager for C++.	\

Continued on next page

Author’s address:

Build System	Description	Default Command
Make	A tool that controls the generation of executables and other non-source files of a program from the program's source files.	make
Meson	An open source build system meant to be both extremely fast, and, even more importantly, as user friendly as possible.	meson setup builddir && cd builddir && meson compile
Ninja	A small build system with a focus on speed.	ninja
Sccache	A ccache-like compiler caching tool which is used as a compiler wrapper and avoids compilation when possible, storing cached results either on local disk or in one of several cloud storage backends.	\
Scons	A software construction tool configured with a Python script.	scons
Sconsolidator	Scons build system integration for Eclipse CDT.	\
Spack	A flexible package manager that supports multiple versions, configurations, platforms, and compilers.	spack install <package_name>
SW	A cross-platform C++ (and other langs) Build System and Package Manager with a lot of packages available.	
tundra	A high-performance code build system designed to give the best possible incremental build times even for very large software projects.	tundra
tup	A file-based build system which inputs a list of file changes and a directed acyclic graph (DAG), then processes the DAG to execute the appropriate commands required to update dependent files.	tup
Premake	A tool configured with a Lua script to generate project files for Visual Studio, GNU Make, Xcode, Code::Blocks, and more across Windows, Mac OS X, and Linux.	premake5 gmake && make config=release
Vcpkg	C++ library manager for Windows, Linux, and MacOS.	vcpkg install <package_name>
waf	Python-based framework for configuring, compiling, and installing applications.	./waf configure && ./waf build
XMake	A lightweight cross-platform build utility based on Lua.	xmake

2 PROMPTS IN CXXCRAFTER

2.1 Parsing build system information

Based on file matching patterns and LLM inference, CXXCrafter extracts build system information, including build system name, version requirement, and build entry location.

DEFINITION 1: BUILD_ENTRY_FILES

```
BUILD_FILES = {
    'Make': 'Makefile',
    'CMake': 'CMakeLists.txt',
    'Autotools': ['configure', 'configure.in',
    'configure.ac', 'Makefile.am'],
    'Ninja': 'build.ninja',
    'Meson': 'meson.build',
    'Bazel': ['BUILD', 'BUILD.bazel'],
    'Xmake': 'xmake.lua',
    'Build2': 'manifest',
    'Python': 'setup.py',
    'Vcpkg': 'vcpkg.json',
    'Shell': 'build.sh'
}
```

The parser defines the `BUILD_FILES` dictionary, which maps common build systems to respective build entry files. The parsing module retrieves build entry files from project directory, and records build systems candidates in `BUILD_SYSTEM_DICT`, along with build entry file locations.

PROMPT 1: Getting the build system and the entry file

I am working on the project `PROJECT_NAME`, which can be built using one of the identified build systems listed below.
Each build system includes an associated entry file : `BUILD_SYSTEM_DICT`
Please select the most appropriate build system and its corresponding entry file that is most likely to successfully build the entire project. The output should be formatted as a tuple like ”’`json (build_system_name, entry_file_path)`”’.
Do not include any additional output.

The parser obtains `BUILD_SYSTEM_DICT` by heuristically filtering the build system candidates, and then allows the LLM to make the final decision.

2.2 Extracting documentation information

The parsing module collects documentation by matching commonly used names of build instruction files. The LLM then chooses from the files, filtering out irrelevant ones.

DEFINITION 2: FILE_PATTERNS

```
FILE_PATTERNS = [
r'^README(\.md|\.txt)?$',
r'^INSTALL(\.md|\.txt)?$',
r'^requirements\.txt$',
r'^\.env$',
r'^config\.yaml$',
r'^install\.sh$',
r'^setup\.py$',
r'^docs/installation\.md$',
r'^docs/setup\.md$',
]
```

The `FILE_PATTERNS` list includes naming patterns for common build specification documents. The parsing module matches documents based on these patterns and forms `DOC_LIST`.

PROMPT 2: Extracting useful document information

I need to build the project `PROJECT_NAME`, and the following documents have been identified as potential resources for this task: `DOC_LIST`.
Please select the most relevant documents that are most likely to assist in successfully building the project itself.
The output should be formatted as a list like ”’`json ['document_file1', 'document_file2', ...]`”’.
Do not include any additional output.

The parser gets `DOC_LIST` by `DEFINITION 2` and asks the LLM to choose from the list.

2.3 Dockerfile generation

Based on the Dockerfile template and the information retrieved by environment parsing, the LLM infers the project’s build process and generates the initial Dockerfile.

DEFINITION 3: DOCKERFILE_TEMPLATE

```
DOCKERFILE_TEMPLATE = """
FROM ubuntu:{ubuntu_version}
ENV DEBIAN_FRONTEND=noninteractive

# Install necessary packages
Run apt-get update
Run apt-get install -y build-essential
Run apt-get install -y software-properties-common

# Install Dependencies
Run apt-get install -y {dependency1}
Run apt-get install -y {dependency2}
...

# Build the project with {build system}
...
"""
```

The `DOCKERFILE_TEMPLATE` serves as a guide to avoid uncertainty in Dockerfile generation process.

PROMPT 3: Getting the initial Dockerfile

Please generate dockerfile which build the project `PROJECT_NAME` from source code according to the dockerfile template `DOCKERFILE_TEMPLATE`.
The source code is located at `PROJECT_PATH`. Move it to the docker container temp directory and build.
Requirements:
1. Install commands must be executed one at a time.
2. Avoid repeating identical RUN commands.
3. Please adhere to Dockerfile syntax. For example, ensure that comments and commands are on separate lines. Comments should start with a # and be placed independently of commands.
Some useful information:
User intention: `USER_INTENTION`
Environment requirement: `ENVIRONMENT_REQUIREMENT`
Docs: `DOCS`
Potential Dependencies (skip installation if useless): `POTENTIAL_DEPENDENCY`

The generation module aggregates all collected information and guides the LLM in constructing the initial Dockerfile.
`DOCKERFILE_TEMPLATE` : a standardized dockerfile format, detailed in `Definition 3` above.
`USER_INTENTION` : certain build requirements specified by the user.
`ENVIRONMENT_REQUIREMENT` : build system information detected in the project.
`DOCS` : document information extracted by `PROMPT 2`.
`POTENTIAL_DEPENDENCY` : dependency information produced by ccscanner.

2.4 Dockerfile modification

If build errors are detected during execution, the error information will be returned to the LLM. The LLM then modifies the Dockerfile based on the previously generated content and the error messages.

PROMPT 5: Modifying the Dockerfile (System message)
Solve the problem according to the error message and modify the dockerfile.
The dockerfile is:\n{last_dockerfile_content}\n
The error message is:\n{feedback_message}\n
Additionally, take note of the following items:
1. If the error message indicates a network issue, do not make any modifications to the Dockerfile.
2. Please return a complete dockerfile rather than just providing advice.
3. Try to keep the beginning of the Dockerfile unchanged and make minimal modifications towards the end of the file.
4. In the dockerfile, commands must be executed one at a time.
5. If some unnecessary modules, such as the testing module, are causing issues, they should be disabled through build options.
6. If required packages, tools, or dependencies are missing, proceed with installing them rather than just verifying their presence.
7. In case errors arise due to specific dependency versions, attempt to acquire and install the exact version of the software that is required.
8. If a 404 error occurs while attempting to download a specific dependency version, verify the correctness of the download link and make any necessary corrections.

Before modifying the Dockerfile, the generation module provides overall guidance and considerations for the LLM.

PROMPT 6: Modifying the Dockerfile (User message)

Content of the last dockerfile:

```
FROM ubuntu:{ubuntu_version}
ENV DEBIAN_FRONTEND=noninteractive

# Install necessary packages
Run apt-get update
Run apt-get install -y build-essential
Run apt-get install -y software-properties-common

# Install Dependencies
Run apt-get install -y {dependency1}
Run apt-get install -y {dependency2}
...

# Build the project with {build system}
...
```

Error message:

```
Error at CMakeLists.txt:66 (find_package):
Could not find a package configuration file provided by "LLVM" with any of
the following names:
...
```

The LLM modifies the Dockerfile based on the original content and the corresponding error message.

2.5 Further verification of the final build success

The execution module runs the Dockerfile. If no build errors are detected, the LLM verifies whether the current Dockerfile can successfully complete the project build to ensure a reliable outcome.

PROMPT 4: Confirming building success

Please verify if the Dockerfile successfully builds the project by examining both the Dockerfile and its output execution messages to confirm the success of the build. If the build is not successful, provide the reason and advice on how to modify it.

Inputs:

- Dockerfile content: `DOCKERFILE`
- Output message: `MESSAGE`

Outputs:

Return a JSON tuple with two elements. The first element is a boolean indicating whether the build was successful. The second element is a string with the reason and advice if the build was not successful (or None if the build was successful).

The output format should be: “‘`json (True, None)`“‘ or “‘`json (False, "<Reason and Advice>")`“‘.

If the build is successful, simply return “‘`json(True, None)`“‘.

If the build fails, return “‘`json(False, <Reason and Advice>)`“‘.

In the execution module, the LLM determines whether the Dockerfile succesfully build the project and gives advice on Dockerfile modification.

`DOCKERFILE`: dockerfile content created by the generation module.

`MESSAGE` : building information returned by docker.