Automatic Fixing of Missing Dependency Errors

Jun Lyu
Nanjing University
Nanjing, China
lvjun@smail.nju.edu.cn

He Zhang[*]
Nanjing University
Nanjing, China
hezhang@nju.edu.cn

Lanxin Yang
Nanjing University
Nanjing, China
lxyang@nju.edu.cn

Yue Li
Nanjing University
Nanjing, China
yueli.dom@outlook.com

Chenxing Zhong[*]
Nanjing University of Science and Technology
Nanjing, China
chenxingzhong@njust.edu.cn

Manuel Rigger
National University of Singapore
Singapore
rigger@nus.edu.sg

***Abstract*—Many build systems, such as Make, rely on build scripts that are written by users to specify dependencies. As a serious dependency error in Makefiles, Missing Dependencies (MDs) can result in compiling and linking outdated artifacts in incremental builds, preventing software project updates from being applied correctly. Many studies have explored the detection of MDs. Automatically fixing those missing build dependency errors has become an apparent but challenging task. The challenges mainly result from Makefiles having complex semantics and project maintainers declaring dependencies in a variety of ways. To address these challenges, we propose a new approach to fixing MDs called MDfixer. The core idea of MDfixer is to identify the dependency declaration style in a Makefile and generate patches for the same declaration style based on declaration graphs and automatic prompt generation. Specifically, MDfixer locates dependency declarations for targets that have errors in the Makefile based on error reports, and then builds a declaration graph for each build target with errors and identifies the target's declaration style based on a distance metric between the target and the dependencies. Based on the declaration graph and automatic prompt generation, MDfixer generates patches with the same style for the dependencies that need to be added. We evaluated the effectiveness and efficiency of MDfixer with 35 well-known projects. The evaluation results show that MDfixer can fix all MDs. We submitted fixes for 2,786 individual dependency issues across 17 projects, with 11 of them merging our pull requests, resulting in a total of 2,099 errors being fixed. MDfixer consumes an average time of 3.31 min for fixing a project, with a median of 62.999s. It can assist practitioners in the effective and efficient fixing of MDs.**

***Index Terms*—Build Tool, Build System Maintenance, Build Dependency Errors, Automatic Program Repair**

I. Introduction

Modern software projects rely on automatic build systems to manage software builds. Build scripts are an integral part of many build systems, such as GNU Make [1], SCons [2], Ant [3], Maven [4], and Gradle [5]. Build scripts specify dependencies, which are source files or artifacts built from source files. Developing such build scripts can be difficult and error-prone, especially for large and complex projects. Although automatic script generators such as Autotools [6] and CMake [7] are helpful, it is still necessary to manually enumerate build dependencies for custom software, which is an error-prone process for large-scale projects. Particularly in C-based projects, 52.68% of the build errors are related to dependency problems [8]. According to McIntosh et al., up to 89% of developers are negatively impacted by build maintenance [9].

Missing Dependencies (MDs) constitute a critical build error in C projects, occurring when users omit necessary file dependencies declared in Makefiles [10], [11]. These omissions compromise incremental build correctness by allowing outdated artifacts to persist, despite successful build status reports [12], [13]. The reason for this is build tools like GNU Make [1] relying on timestamp comparisons: MDs prevent updated dependencies (*e.g.*, modified header files) from triggering necessary rebuilds. As demonstrated in Fig. 1 with the *clib* project, when a header file (src/version.h) changes but is not declared, dependent binaries ("BINS") may link obsolete versions without build failures, causing silent runtime errors. This discrepancy between declared and actual dependencies [14] creates two critical risks: (1) undetected build inaccuracies where outputs do not reflect recent code changes, and (2) subsequent test failures from inconsistent artifact versions. The compounding effect makes MDs resolution both urgent and non-trivial in practice.

While significant progress has been made in detecting Makefile dependency issues through static analysis and dynamic tracing techniques [15], [10], [11], [16], [13], the automated remediation of MDs remains an unresolved challenge in build system maintenance. Current state-of-the-art repair methodologies predominantly target build failures [17], [18], [19], [20], [21], [22]—scenarios where dependency errors manifest as immediate compilation interrupts or fatal runtime exceptions. These approaches leverage explicit error signals from build logs to guide fixes, a paradigm inapplicable to MDs resolution as MDs typically permit successful builds while silently producing outdated or inconsistent artifacts [10], [11], [16].

The automatic fixing of Makefiles presents significant challenges, primarily stemming from two factors: 1) different ways of declaring dependencies such as explicit declarations

[*] Corresponding author: He Zhang and Chenxing Zhong.

```
1.    SRC = $(wildcard src/*.c)
2.    COMMON_SRC = $(wildcard src/common/*.c)
3.    DEPS = $(wildcard deps/*/*.c)
4.    OBJS = $(DEPS:.c=.o)
5.    $(BINS): $(SRC) $( COMMON _SRC) $(MAKEFILES) $(OBJS)
        $(CC) $(CFLAGS) -o $@ $(COMMON_SRC) src/$(@:.exe=).c
        $(OBJS) $(LDFLAGS)
```

Fig. 1: MDs from the *Clib* Project (Commit ID: 2e8f12f)

and implicit declarations, where explicit declarations involve macro expansions, and target-specific rules, while implicit declarations rely on GNU Make's default inference (*e.g.*, pattern rules, suffix rules) that are rarely documented; and 2) projects using different declaration styles. As shown in Fig 1, the explicit declarations (lines 1–3) explicitly enumerate specific file paths:
SRC = $(wildcard src/.c) $\rightarrow$ *src/clib.c*

These require explicit declarations because the build system cannot predict where to find application sources (`src/`) versus third-party dependencies (`deps/`). In contrast, line 4's implicit declaration leverages Make's default rules for compiling `.c` files to `.o` objects:
OBJS = $(DEPS:.c=.o) $\rightarrow$ *deps/list/list.c deps/list/list.o*

This works because Make automatically applies its built-in pattern rule `%.o: %.c` when no explicit declaration exists. However, such implicit declarations have limitations in dependency tracking. If "*deps/list/list.c*" includes "*deps/list/list.h*", the implicit declarations do not automatically detect changes in the header file. For example, modifying "*deps/list/list.h*" would not trigger recompilation of "*deps/list/list.o*". If an implicit declaration omits critical dependencies (*e.g.*, missing "*deps/list/list.h*"), developers cannot directly "patch" the declaration. For example, attempting to fix this by adding "*deps/list/list.h*" to line 4's command list will fail, because implicit declarations are hard-coded in Make and ignore user modifications [1]. Only explicit declarations allow incremental fixes via dependency appending.

Effective automatic patch generation for Makefiles requires adaptive handling of diverse dependency declaration styles—the syntactic patterns developers use to specify target dependencies. As illustrated in Fig. 2, the *libco* project demonstrates coexisting styles: explicit dependency enumeration (line 2, "colib") versus macro-based abstraction (line 3, "libcolib.a"). These stylistic variations necessitate context-aware fixes that align with original patterns rather than imposing uniformity. We define three declaration styles, and the main difference between them lies in how the declarations use dependencies and macros. Macros are essentially predefined code snippets, often representing column filenames (in this paper, we call them items) in a Makefile [1]. Consequently, when MDs exist in implicit declarations, we cannot simply add missing dependencies to the declarations, thereby posing significant challenges for effective issue resolution. For example, in an implicit declaration using a rule like `%.o: %.c` with macros (*e.g.*, SRC_DIR = src, HEADERS = $(SRC_DIR)/*.h), manually appending $(HEADERS) as a dependency to the rule may be ignored by Make. This is because Make's rules

```
1.    COLIB_OBJS = co_epoll.o co_routine.o co_hook_sys_call.o
      coctx_swap.o coctx.o co_comm.o
2.    colib: libcolib.a libcolib.so
3.    libcolib.a: $(COLIB_OBJS) $(ARSTATICLIB)
```

Fig. 2: Declarations from the *Libco* Makefile (Commit ID: dc6aafc)

```
1.  Modules/_decimal/libmpdec/basearith.o:
      $(srcdir)/Modules/_decimal/libmpdec/basearith.c
                $(LIBMPDEC_HEADERS) $(PYTHON_HEADERS)
                $(CC) -c $(LIBMPDEC_CFLAGS) -o $@
                 $(srcdir)/Modules/_decimal/libmpdec/basearith.c
```

Fig. 3: Declarations from the *CPython* Makefile (Commit ID: 1dce007)

(%) only manage dependencies between the pattern-matched source and target files, not additional, non-matched files like $(HEADERS). Thus, successful repair demands style-specific fixing to ensure syntactic consistency between original and patched code.

To address these challenges, this paper proposes the first automatic fixing approach called MDfixer for MDs of C-based and Make-based projects. Our core idea is that dependency errors in explicit and implicit declarations stem from fundamental differences in build logic, requiring different fix strategies. Distinct fix strategies are required because explicit and implicit declarations in Makefiles are different, necessitating distinct fix strategies. Explicit declarations provide concrete, statically analyzable rules where missing dependencies can be directly added to the declaration itself. Conversely, implicit declarations rely on GNU Make's pattern rules and suffix rules; these are templates that dynamically generate dependencies during build execution. Consequently, the traditional repair strategy of simply appending missing dependencies to a declaration is inherently unsuitable for errors originating from implicit rules, as they lack a fixed, modifiable target-dependency specification. Specifically, MDFixer uses a different approach depending on whether the Makefile uses an explicit or implicit declaration. MDfixer first build a declaration graph for each build target, based on the graph, identifies whether the error exists in explicit declarations or implicit declarations. For explicit declarations, MDfixer computes the distance from each dependency in the graph to the target and determines the target's declaration style based on that distance. MDfixer then generates patches for targets with MDs using multiple patch generation strategies. For implicit declarations, MDfixer adds dynamic dependency tracking files (*e.g.*, ".d" files [1]) by automatically generating prompts instructing LLMs to modify implicit declarations. Dynamic dependency tracking files allow Make to correctly perform incremental builds [1], and it finishes fixing MDs without destroying the dependency style of the original declarations.

We evaluate the effectiveness and efficiency of MDfixer on 35 projects. As EChecker [12] is regarded as the state-of-the-art method of detecting build dependency errors in Makefiles, the detection results of MDs by EChecker are used as the ground truth. We determine whether MDfixer has completed the fixing of MDs by comparing the results of EChecker's detection of MDs before and after the project has been fixed. The evaluation results show that MDfixer fixed all MDs in our targets, with a median time consumption of 62.999s and an average of 3.31 min. In addition, we submitted 17 individual pull requests to the 17 projects, of which 11 projects merged our pull requests, resulting in an overall 2,099 MDs being fixed in these projects. The effective and efficient fixes of MDs by MDfixer relieve developers from the considerable burden of debugging and correcting incorrect incremental builds caused by MDs. The core idea of MDfixer can be applied to other high-level build tools (*e.g.*, CMake), but the patch generation component needs to be changed to match the syntax of other high-level tools.

II. Preliminaries

This section introduces important concepts related to the approach presented in this paper, *e.g.*, *declaration styles*, *declaration graphs*, and *error reports*.

At a high level, a Makefile comprises target-dependency pairs (t_i, d_i) where t_i specifies a build objective, and d_i denotes its prerequisite set $d_{i,1}, \dots, d_{i,n}$. Dependencies $d_{i,j}$ may manifest as concrete files, targets, or macros (m_k). Each macro m_k encapsulate groups of files or targets.

We refer to the declaration style as the way the dependencies are declared. In other words, a declaration style refers to how the dependencies $d_{i,j}$ are structured. Practitioners might use specific dependencies or macros m_k for each declaration when developing a Makefile.

$$\text{Target} = \text{Dependency}_1,\ \dots,\ \text{Dependency}_n \tag{1}$$
$$\text{Target} = \text{m}_k \tag{2}$$
$$\text{Target} = \text{Dependency}_1,\ \dots,\ \text{Dependency}_n, \text{m}_k \tag{3}$$

a) Three Declaration Styles: We define these styles as divided into three main categories. The main difference between styles is whether macros are used in the declaration. The first style is to list all atomic dependencies in the declaration, which is structured as shown in Equation (1). For example, in Fig. 2, practitioners list the dependencies “libcolib.a” and “libcolib.so” for the target “colib” (line 2). The second style is to use only macros in the declaration, which is structured as shown in Equation (2). For example, in Fig. 2, practitioners first declare a macro “COLIB_OBJS” (line 1), after declaring it as a dependency of the target “libcolib.a” (line 3). The third style is to mix macros and list dependencies in the declaration, which is structured as shown in Equation (3). For example, in Fig. 3, practitioners declare a Atomic Dependency “Modules/_decimal/libmpdec/basearith.c” and a macro “LIBMPDEC_HEADERS” as dependencies of the target “Modules/_decimal/libmpdec/basearith.o”.

$$\text{Target} = \text{Dependency}_n \tag{4}$$
$$\text{Target} = \text{Dependency}_n, \text{Added\_Dependency}_i \tag{5}$$

b) Declaration Graph: The declaration graph consists of all n pairs (t_i, d_i), in which each target and dependency of a pair can be seen as nodes, with an edge from the dependencies to their target.

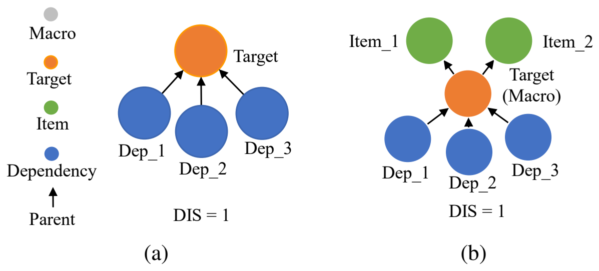


Fig. 4: Equation (1) Declaration Graph

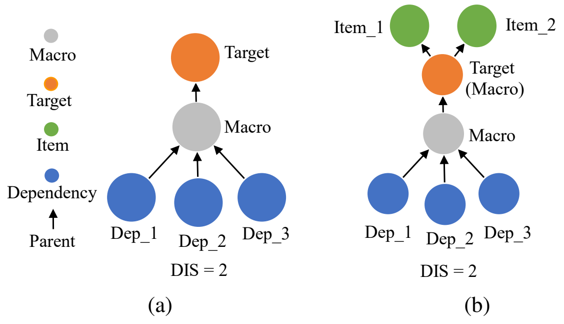


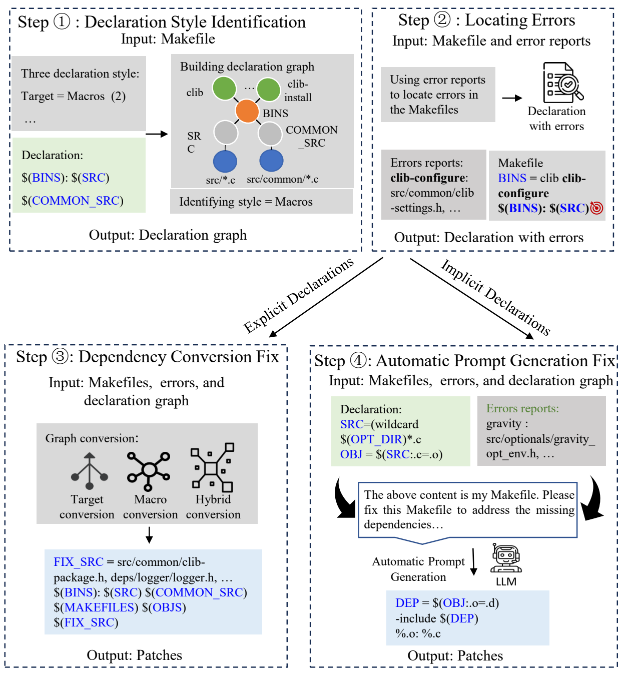
Fig. 5: Equation (2) Declaration Graph

Fig. 4 and Fig. 5 show examples of declaration graphs. Fig. 4a and Fig. 4b are examples of Equation (1), Fig. 5a and Fig. 5b are examples of Equation (2). Examples of Equation (3) are combinations of the above figures that are not shown here. In the declaration graph, when the macro is a target, the macro acts as an intermediate node, with the declared items in the macro as its parents and the macro’s dependencies as its children (Fig. 4b and Fig. 5b). When the macro is a dependency, the macro acts as an intermediate node, its parent is the target, and its children are the declarations in the macro (Fig. 5a).

c) Dynamic dependency tracking files (.d): This file is mainly used to record dependencies between source files, such as header file inclusion relationships. It is used to help GNU Make determine which files need to be recompiled during subsequent compilation.

d) Error reports: Our approach aims to fix MDs in Makefiles, its inputs are project repositories and dependency error reports, and its outputs are fixed Makefiles. The dependency error reports used by our approach come from the state-of-the-art MDs detector—EChecker [12], which contains a number of i pairs (t_i, e_i) where t_i denotes the target in the Makefiles, and e_i of a list of MDs. Our approach is not exclusive to EChecker. It can be integrated with other dependency error detection tools as well [10], [16].

e) MDs fixing problems: Our approach generates patches according to the declaration styles. The fixes by our approach do not violate the original declaration style, it follows the original style in the Makefile to generate patches instead of generating a new Makefile, which may result in the project maintainer’s extra burdens, such as potentially increased maintenance costs. Typically, a build target in a Makefile declares



Step ① : Declaration Style Identification
Input: Makefile
Three declaration style:
Target = Macros (2)
…
Declaration:
$(BINS): $(SRC)
$(COMMON_SRC)
Building declaration graph
clib
…
clib-install
BINS
SRC
COMMON_SRC
src/*.c
src/common/*.c
Identifying style = Macros
Output: Declaration graph
Step ② : Locating Errors
Input: Makefile and error reports
Using error reports to locate errors in the Makefiles
Declaration with errors
Errors reports:
clib-configure: src/common/clib-settings.h, …
Makefile
BINS = clib clib-configure
$(BINS): $(SRC)
Output: Declaration with errors
Explicit Declarations
Implicit Declarations
Step ③: Dependency Conversion Fix
Input: Makefiles, errors, and declaration graph
Graph conversion:
Target conversion
Macro conversion
Hybrid conversion
FIX_SRC = src/common/clib-package.h, deps/logger/logger.h, …
$(BINS): $(SRC) $(COMMON_SRC)
$(MAKEFILES) $(OBJS)
$(FIX_SRC)
Output: Patches
Step ④: Automatic Prompt Generation Fix
Input: Makefiles, errors, and declaration graph
Declaration:
SRC=(wildcard $(OPT_DIR)*.c
OBJ = $(SRC:.c=.o)
Errors reports:
gravity : src/optionals/gravity_opt_env.h, …
The above content is my Makefile. Please fix this Makefile to address the missing dependencies…
Automatic Prompt Generation
LLM
DEP = $(OBJ:.o=.d)
-include $(DEP)
%.o: %.c
Output: Patches

Fig. 6: Overview of MDfixer

one or multiple Dependency$_n$ (Equation (4)). MDfixer fixes MDs present in the target by adding the Added_Dependency$_i$ (Equation (5)).

III. Approach

Our key insight for solving MDs is that explicit and implicit declarations originate from fundamentally different build logics. Implicit and explicit declarations represent different challenges in terms of fixing, namely, semantic gaps in context patterns and syntactic omissions in explicit declarations. Based on this, we propose two complementary fixing strategies: style-aware fixing for explicit declarations, and an LLM-based approach for implicit declarations.

A. Overview

MDfixer preprocesses the input Makefile, identifying all build targets and macros in the Makefile. To determine the declaration style of these targets, MDfixer builds a declaration graph and calculates the Declaration Distance Score (DIS)—a metric measuring the path distance between dependencies and targets in a declaration graph (step ①). Based on the error reports, MDfixer locates and determines which declarations of the targets need to be fixed (step ②). For patch generation (steps ③ and ④), MDfixer deploys a hybrid strategy: *dependence conversion* for explicit declarations, while the fixes of implicit declarations leverage an LLM-based approach. Having two fixing strategies ensures syntactic non-invasiveness (not altering original rule structures) while achieving dependency semantic completeness. Since explicit declarations have a static syntactic structure that enables deterministic dependency insertion, we prioritize rule-based methods. However, for

```
1. PROGS = colib example_poll example_echosvr example_echocli
   example_thread  example_cond…
2. all:$(PROGS)
3. example_echosvr: example_echosvr.o $(BUILDEXE)
4. Error report example_echosvr.o: libco/co_routine.h
```

Fig. 7: Calculating DIS in *Libco* Makefile

implicit declarations, rule-based methods become ineffective because their dependencies are encoded in dynamic, project-specific logic (*e.g.*, pattern rules) and lack a unified syntax. In such cases, LLMs can identify and fix implicit MDs by analyzing semantic patterns across different projects, thereby overcoming the limitations of rule-based methods.

B. Declaration Style Identification

MDfixer identifies declaration styles in Makefiles using declaration graphs (step ①).

MDfixer first parses the Makefile and builds a declaration graph for each target and identifies the declaration style using the graph. MDfixer first identifies all macros and their corresponding items to build the declaration graphs and subsequently locate errors. MDfixer identifies the declaration style by calculating the DIS between the dependencies and the target node in the declaration graph. DIS describes the distance from the dependency to the target in the declaration graph, and it can accurately reflect the declaration structure of the build target, *i.e.*, how atomic dependencies and macros are used in the declaration of this target. The relationship between declaration style and DIS is shown in Equation (6). Equation (1) corresponds to all dependent nodes having a DIS to the target of only 1, while Equation (2) corresponds to DIS greater than or equal to 2. Equation (3) corresponds to the existence of both nodes with a DIS equal to 1 and nodes with a DIS greater than or equal to 2. Equation (1) states that the target declarations are all atomic dependencies, so DIS is all 1. Equation (2) states that the target declarations are all macros, so DIS is all 2. Meanwhile, Equation (3) states that the target uses both atomic dependencies and macros, so DIS is both 1 and 2. MDfixer classifies any DIS that does not belong to these three categories as implicit declarations

According to the error report (See Fig 7), the target "example_echosvr.o" contains MDs. MDfixer attempts to locate the declaration of "example_echosvr.o" and calculate the DIS. MDfixer finds that "example_echosvr.o" appears as a dependency on line 2, but fails to find the declaration of "example_echosvr.o" as a target. Therefore, MDfixer does not calculate the DIS for "example_echosvr.o" and classifies it as "other" according to equation 6. MDfixer identifies that "example_echosvr.o" uses an implicit declaration.

$$Style = \begin{cases} DIS = 1, \ Equation\ (1) \\ DIS \geq 2, \ Equation\ (2) \\ DIS = 1 \ and \ DIS \geq 2, \ Equation\ (3) \\ Other, \ Implicit\ declarations \end{cases} \tag{6}$$

C. Locating Errors

This subsection describes how MDfixer locates errors (step ②). Numerous methods for detecting MDs and generating error reports are available [10], [16], and our approach is compatible with them.

State-of-the-art tools for detecting build dependency errors [12], [23] rely on GNU Make to interpret build scripts. They extract each build target and its dependencies by analyzing the GNU Make internal database. However, public dependency error detection tools [10], [16], [12], [23] typically handle specific build target types and error formats in their reports and do not pinpoint the exact location of errors in Makefiles. Furthermore, build target declarations in Makefiles can be implicit; the extensive use of macros complicates error localization.

To address this, MDfixer first identifies build targets with errors in each Makefile. It parses macros to resolve those associated with build targets, enabling precise identification of MD-containing declarations. When macros are used to represent build targets, MDfixer checks for MDs in the logic underlying these macros. Finally, MDfixer locates erroneous declarations through a line-by-line analysis of the Makefile.

We use an example to explain how MDfixer locates macros containing MDs (see Fig 8) MDfixer first parses the macro “BINS” to obtain a series of specific targets represented by “BINS” (line 1). Next, MDfixer searches for the declaration corresponding to “clib-configure” based on the error report. Based on the previous macro parsing, MDfixer finds that “clib-configure” corresponds to “BINS”, and therefore locates the declaration corresponding to “BINS” (line 5). Here, MDfixer completes the location of dependency errors contained in the macro.

D. Generating Patches for Explicit Declarations

MDfixer employs a declaration graph-driven approach to differentiate between explicit and implicit error declarations (steps ③ and ④), implementing different tailored fix strategies. For explicit declarations, building upon the error localization results from previous stages (step ③), the *style-aware fixing* methodology processes erroneous declarations by converting missing dependencies (MDs) into syntax-compliant representations that match existing declaration patterns in Makefiles. This conversion mechanism dynamically adapts to three fundamental declaration styles observed in build scripts.

The conversion process begins by analyzing error reports (t_i, e_i) from detection tools such as EChecker, where t_i denotes the problematic build target and e_i represents missing dependencies. For example, if the declaration style is Equation (2), *style-aware fixing* will convert the dependencies into macros, then it uses macros to fix errors. *Style-aware fixing* uses different conversion methods depending on the declaration style of the corresponding declaration of t_i in the Makefile, including *target conversion*, *macro conversion*, and *hybrid conversion*.

Target conversion processes the declaration style of Equation (1). MDfixer performs *Target conversion* by directly appending the dependencies e_i to t_i's prerequisite list. Specifically, *Target conversion* sequentially extracts dependencies associated with the same target from the build dependency error reports and adds them one by one to the corresponding declarations in the Makefile. This ensures that the modified declarations preserve original Makefile conventions.

Macro conversion processes the declaration style of the Equation (2), which converts dependencies into macros. In *macro conversion*, MDfixer converts e_i to a macro, and then adds the macro to the corresponding declaration of t_i in the Makefile. Specifically, *Macro conversion* sequentially extracts dependencies belonging to the same target from the build dependency error reports and declares these dependencies as different macros. MDfixer then inserts these macros into the target's declarations in the Makefile to complete the fix.

Hybrid conversion processes the declaration style of the Equation (3). When handling dependencies from unified directories (*e.g.*, src/common/*.h), the system deploys wildcard macros under the conditions: 1) All same-type files in the directory must constitute valid dependencies; 2) There are no extraneous files of the specified type. This prevents over-inclusion while maximizing declaration efficiency. For cross-directory dependencies or heterogeneous file types, MDfixer reverts to target conversion with explicit enumeration, as exemplified in Fig 8 where “BINS” target dependencies combine macro-abstracted local resources and explicitly listed external components. The framework enforces conversion rules through a macro generator that tracks dependency origins, across all fix operations.

```
1.  BINS = clib clib-install clib-search clib-init clib-configure clib-build
    clib-update clib-upgrade clib-uninstall
2.  SRC = $(wildcard src/*.c)
3.  COMMON_SRC = $(wildcard src/common/*.c)
4.  all: $(BINS)
5.  - $(BINS): $(SRC) $(MAKEFILES) $(OBJS)
6.  + $(BINS): $(SRC) $(COMMON_SRC) $(MAKEFILES) $(OBJS)
7.  $(CC) $(CFLAGS) -o $@ $(COMMON_SRC) src/$(@:.exe=).c
    $(OBJS) $(LDFLAGS)
# EChecker errors reports:
8.  clib-configure:  src/common/clib-settings.h, src/common/clib-settings.c
```

Fig. 8: Fixed Makefile and MDs from the *Clib* Project (commit ID: 2e8f12f)

E. Generating Patches for Implicit Declarations

MDfixer generates patches for implicit declarations using LLMs by automatic prompt generation (step ③).

Fixing missing dependencies caused by implicit declarations presents unique challenges, as these dependencies are highly project-specific contextual logic, such as different pattern rules. These constructs introduce heterogeneity across Makefiles. The variations in implicit declarations across different Makefiles are significant, making standardized repair methods impractical.

LLMs offer a solution by leveraging their ability to understand the complex semantics and contextual patterns within specific Makefiles. This deep understanding enables LLMs to accurately identify the specific implicit rules that require

```
1.  {Makefile}
2.  The above is my Makefile. Please fix this Makefile to address the
    missing dependencies. Please apply a fix that is concise and
    compact. Whenever possible, consider not breaking the
    declaration style of the original Makefile when fixing it.
3.  If macros are used, the fix is performed using macros. If the target
    lists dependencies, you list dependencies. For implicit declarations
    (e.g., %.o: %.c) consider using .d for the fix.
4.  The structure of the following error report is that there is a target
    with a missing dependency: it is missing the declared dependency.
5.  The missing dependencies are as follows:
6.  {unfixed_targets}
```

Fig. 9: Automatic Prompt Generation for Implicit Declarations

repair, followed by employing various strategies such as modifying pattern prerequisites, adjusting build instructions, or adding additional rules to accommodate the unique structures of different projects, rather than simply appending dependencies. MDfixer automatically generates prompts based on the error report and Makefile, and then instructs the LLMs to fix it. The input for this step comes from the Makefiles and declarations with errors from steps ① and ②.

a) Prompts format: The essence of these prompts is to transform the information from steps ① and ② into concise instructions and relevant inputs (*i.e.*, Makefile or declaration graphs, and unfixed errors). An example prompt is shown in Fig. 9, with all prompts available in the supplementary material.

As shown in Fig 9, this prompt is used when MDfixer instructs an LLM (*e.g.*, Deepseek or ChatGPT) to generate a new Makefile (patch). This prompt combines four elements: 1) dependency declarations and macro definitions (line 1), 2) missing dependency listings with target associations (lines 5–6), 3) style-preservation constraints (line 2), and 4) exemplar fixes for analogous error patterns (lines 3–4). This structured approach ensures LLMs generate valid patches while maintaining the original Makefile's abstraction levels and declaration styles.

b) Overcoming input/output limitations of LLMs: MDfixer initiates by processing the complete Makefile, leveraging its holistic semantic context to guide LLM-driven repairs. For large-scale projects, however, verbose Makefiles often exceed LLMs' token limits. When this occurs, MDfixer dynamically transitions to a declaration graph representation, which encapsulates only essential elements (targets, macros, dependencies) while omitting non-critical content (build commands, comments). This reduces input size to fit within LLMs' processing bounds, ensuring efficient analysis of dependency structures (*e.g.*, macros and target relationships) critical for resolving MDs.

IV. Evaluation

We evaluate the effectiveness and efficiency of MDfixer on practical and widely used software systems. The evaluation is driven by the following questions (Q).

Q1: How effective is MDfixer in fixing MDs?

Goal. This question explores the effectiveness of MDfixer in fixing MDs. We sought to determine whether MDs in Makefiles can be fixed correctly.

Method. To answer Q1, we used EChecker to check for dependency errors in the MDfixer-fixed projects, and calculated how many errors MDfixer fixed by comparing the before-fix and after-fix error reports by EChecker. Ideally, MDfixer could fix all MDs, meaning no after-fix MDs reports remain. In addition, to the best of our knowledge, there is no existing method that offers automatic fixing of MDs, therefore we cannot compare MDfixer with other related methods.

Q2: How efficient is MDfixer in fixing MDs?

Goal. This question explores the time consumption of MDfixer in fixing MDs.

Method. To answer Q2, we recorded the time spent on fixing the Makefiles for each project by MDfixer. To eliminate experimental biases resulting from fluctuations in device performance, we executed each fix 5 times, and the final result is calculated as the median.

Q3: What is the fix effectiveness compared to an LLM-only approach, and when varying between different LLMs?

Goal. This question compares the effectiveness of MDfixer's fixes using different LLMs and compares the effectiveness of MDfixer's fixes with those using only LLMs.

Method. MDfixer uses LLMs to fix errors originating from implicit declarations. we first assess MDfixer's efficacy when leveraging different LLMs (GPT-4 and Deepseek-R1) to fix errors in implicit declarations. Additionally, we benchmark standalone LLM performance (using the same models) by directly applying them to fix MDs without MDfixer's structured prompt engineering and dependency graph analysis. For both scenarios, we use Equation (6) to quantify changes in declaration style before and after fixes, ensuring that modifications do not disrupt the original syntax conventions. We consider a successful fix to be one that successfully eliminates MDs and retains the original declaration style.

Q4: Are the artefacts produced before and after the fix identical?

Goal. This question verifies whether MDfixer's fixes can cause an incorrect artifact build. The purpose of MDfixer is to fix MDs in Makefiles, which unavoidably modifies the Makefiles and may affect the build of the software. Therefore, we investigate whether MDfixer's fixes affect the final artifact. If the artifact is changed when initiating a clean build, it is a failed fix. Ideally, MDfixer's fixes do not result in an incorrect artifact build.

Method. To answer Q4, we leverage the deterministic nature of Make-based builds: when dependency graphs remain identical across clean builds, the compilation sequence and output binaries are inherently reproducible. By comparing pre-fix and after-fix dependency graphs, we verify that MDfixer's corrections only augment missing edges without altering legitimate build paths. Suppose the pre- and after-fix dependency graphs are identical. In that case, it indicates that the build system perceives the same set of dependencies and will therefore

TABLE I: Important Information about Subjects and Results of MDfixer Fixing MDs

System	Project	Stars	Commit	Fixed	Sub*	Mer*
GNU Make	CPython	60k	9940093	30	30	0
	kleaver	8	da77621	37	-	-
	clib	4.8k	2e8f12f	49	49	49
	zlib	5.3k	0f51fb4	3	3	3
	Generic-C	6	0b3c533	1	-	-
	fzy	6.1k	eb4cbd4	17	-	-
	tcc	378	255ba0e	4	-	-
	CacheSimulator	5	08a4c52	3	-	-
	iodine	6.4k	50caf2c	42	-	-
	PacVim	3.3k	ca7c883	14	-	-
	sds	5.1k	a9a03bb	1	1	1
	mon	1.1k	9754dc2	4	-	-
	genann	2.1k	4f72209	7	-	-
	pixiewps	1.6k	464326f	97	-	-
	nyancat	1.5k	32fd2eb	3	-	-
	coroutine	2.5k	a626303	1	-	-
	libsass	4.3k	7037f03	2,901	-	-
	jsmn	3.8k	25647e6	8	-	-
	pwnat	3.5k	30b9c79	8	8	8
	fastText	25.6k	1142dc4	35	-	-
	libco	8.1k	dc6aafc	36	-	-
	cppcheck	5.5k	5a05e8d	18	-	-
Autotools	ck	2.3k	07e90d0	216	216	216
	gravity	4.2k	a31e74d	304	304	304
	mpc	2.6k	65f20a1	1	-	-
	coturn	12.8k	24e99ec	10	10	10
	n2n	6.7k	23e5160	184	184	-
	jemalloc	10.3k	1972241	1	1	0
	libhv	7.3k	23ff591	1,161	1,161	1,161
	fio	5.8k	ac2aa2c	298	298	298
	sysstat	3.2k	64cd54e	37	37	37
	unbound	3.8k	85e916e	42	42	0
	liburing	3.3k	81fe3b9	12	12	12
	rsync	3.8k	797e17f	89	89	0
	tcpdump	3k	8231f2b	525	525	0
Sum				6,161	2,786	2,099

-: Projects are not active in the last three months Sub*: Submission Mer*: Merged

execute the same sequence of operations—such as compiling the same source files with the same flags and linking the same libraries—to produce the artifact. We instructed EChecker to output the actual dependency graphs of both and compared the similarity of the two graphs. Ideally, the two graphs are identical. The reason for this is that MDs are dependencies that are not declared in Makefiles, but instead are used during build processes. The build process will not change as MDs are fixed by adding declarations of these dependencies to the Makefiles. In addition, we removed the ".d" file added by MDfixer when calculating the similarity, which was generated additionally and does not affect the builds.

A. Experiment Setting

a) Subjects: We selected 35 projects to evaluate MDfixer. We chose the projects based on the following steps. In the first step, we included all projects by the latest study in which MDs were found [12]. For greater diversity, in the second step, we selected 16 more projects from the other previous study [16]. We identified these 16 projects by removing projects that used build systems other than GNU Make or Autotools, or the latest version of the project that could be detected for MDs by EChecker. To further demonstrate our approach's applicability on Autotools, we selected 10 additional projects using Autotools. First, we randomly selected C/C++ projects from GitHub with over 3,000 stars one by one. From these, we further sampled projects and excluded those that did not use Autotools or for which dependency errors could not be detected. Important information about the subjects is shown in Table I.

b) Approach application methodology: After executing the pre-processing of the project (*e.g.*, `./autogen`, `./configure`, and `./cmake`), all projects are checked for build dependency errors by EChecker. Then, the dependency error reports for each project are used as input to MDfixer.

c) Ground truth: For Q1, we used EChecker's results as a ground truth for whether any errors remained in the projects to be fixed. Each project is checked for MDs by EChecker before and after fixing. A comparison of the two detection results is used to determine whether MDfixer has fixed the error or introduced new errors. Specifically, we compared each error report before and after the fix for each build target, and when a particular error report for this target disappeared from the after-fix detection, we considered that MDfixer successfully fixed the error, and vice versa as a failed fix. In addition, if a new error appears, we consider that MDfixer's fix introduces new MDs.

d) Experiment environment: The experiments were conducted on a Linux server running Ubuntu 22.04 with an Intel(R) Xeon(R) Platinum 8163 CPU@2.50GHz, 96 cores, and 376GB of physical memory. To eliminate experimental bias due to variations in device performance, we performed each experiment five times and took the median time consumption.

B. Effectiveness (Q1)

To answer this question, we evaluated whether MDfixer can successfully fix MDs. The results of fixing errors are shown in Table I. The seven columns show the project name, the sum number of MDs detected by EChecker, the number of MDs, the number of MDs fixed by MDfixer, the number of MDs that remain unfixed, the submission fixes by pull requests, and the merged fixes.

a) Results: MDfixer fixed all MDs (6,161) in the 35 projects. MDfixer completed 100% fixes in 35 projects and did not introduce new MDs for any of the projects. The high effectiveness is due to its fix strategies combining precise error localization with style-aware patch generation. By identifying and parsing all macros in Makefiles, MDfixer gains precise insight into dependency structures, enabling it to pinpoint declarations (explicit or implicit) containing MDs. The tool then analyzes declaration styles to apply targeted repair strategies: for explicit declarations, it uses style-aware fixing to generate style-consistent patches by adapting dependencies to the existing syntax (*e.g.*, target, macro, or hybrid conversion); for implicit declarations, it employs large language models (LLMs) to automatically generate context-aware fixes via structured prompts, ensuring alignment with the original Makefile's syntax and logic. This combination of granular error localization, style-sensitive patching, and adaptive LLM integration allows MDfixer to achieve complete MDs resolution while preserving build integrity across all tested projects.

b) Merged fixes: We submitted fixes to the maintainers of 33 projects by GitHub pull requests (16 projects have been inactive in the last three months, 2 projects have been archived). We received positive feedback from maintainers of 11 projects. The maintainers of the *clib*, *zlib*, *ck*, *sds*, *pwnat*, *gravity*, *libhv*, *liburing*, *coturn*, *fio*, and *sysstat* confirmed all fixes (2,099 fixes) and merged our pull requests. The six fixes based on our issue reports have been confirmed and merged as of the time of writing [24]. The *unbound* maintainers confirmed the fixes, but rejected our PR. They fix the MDs in their own way. Notably, the cppcheck project's maintainers independently fixed 18 MDs in commit afe05d0, and a post-hoc comparison confirmed these fixes matched those generated by MDfixer.

An illustrative example from CPython (Fig 10) demonstrates how MDfixer's approach aligns with but may not always match human maintainers' preferences. The target "xmltok.o" has two errors "xmltok_impl.c" and "xmltok_ns.c". According to the declaration style of "xmltok.o", MDfixer declares a new macro to fix the errors (lines 1 and 2). However, the developers considered that no new macro was needed, hence they added "xmltok_impl.c" and "xmltok_ns.c" to the original macro "LIBEXPAT_HEADERS" to complete the fix (lines 3 and 4). In addition, project maintainers have unique requirements for each Makefile. For example, aesthetic requirements, *CPython* maintainers write all dependencies in multiple lines in alphabetical order. MDfixer fixes may not meet these requirements (*e.g.*, certain style conventions). Despite such adjustments, MDfixer's recommendations served as a basis for addressing the underlying MDs

c) Common patterns: From the 35 subjects, there were 17 subjects with errors in explicit declarations and 19 subjects with errors in implicit declarations, among which Libco had errors in both explicit and implicit declarations. The three fix modes in explicit declarations were dependency list (71%), macro (23%), and hybrid mode (6%).

Answer to Q1: *Within 35 projects, MDfixer can fix all (6,161) MDs without introducing new MDs, which demonstrates its high effectiveness.*

C. Efficiency (Q2)

To evaluate efficiency, we calculated the time consumption of MDfixer on fixing MDs. MDfixer's fix consists of two main stages: style-aware fixing and prompting LLMs, and we record the time consumed by MDfixer in each stage. Each experiment was performed five times, and we calculated the median value of the overall time recorded. The results are shown in Table II.

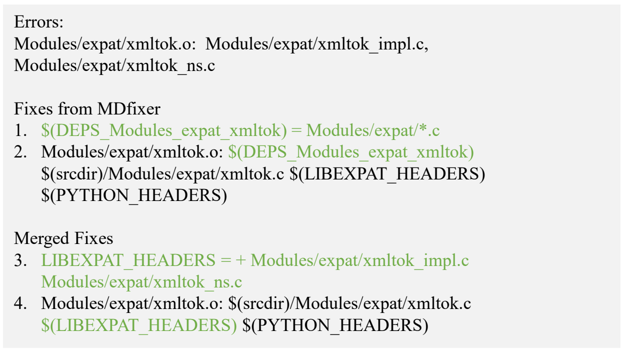
a) Results: MDfixer is efficient at fixing the dependency errors of explicit declarations. In 16 projects, the duration of fixing ranges from 0.005s (fastText) to 0.046s (CPython), with an average total time consumption of 0.016s and a median of 0.008s. This highlights MDfixer's high efficiency, even when dealing with large-scale projects, such as *CPython*. MDfixer uses LLMs when fixing MDs originating from implicit declarations, which is therefore more time-consuming. The time consumed by MDfixer to fix MDs ranged from 788.313s (*unbound*) to 56.653s (*coroutine*) across the 15 projects. Here, the main time consumption of MDfixer comes from communicating with LLMs. Overall, MDfixer consumed an average of 198.386 to fix MDs, with a median of 62.999s.

Answer to Q2: *With 35 fixed projects, MDfixer fixes MDs in an average of 198.386, and a median of 62.999s, which demonstrates its efficiency.*

D. Effectiveness of Components (Q3)

To answer this question, we designed two experiments. First, we let MDfixer use Deepseek and ChatGPT to compare the MDfixer fix results using different LLMs. Second, we use the same prompts but only use LLMs to fix MDs, and we compare the fix results of LLMs and MDfixer. We execute each experiment five times, compare the output results of LLMs each time, and take the output with the highest text similarity as the final result. The results are shown in Table II.

a) Use Different LLMs: MDfixer was able to fix all the errors using Deepseek, whereas it was only able to fix 24 projects using ChatGPT (with a fix success rate of only 68.57%). The main reason MDfixer fails to fix using ChatGPT is that ChatGPT tends to fix errors by listing all dependencies, which breaks the original declaration style. As shown in Fig 11, the maintainers of PacVim use macros in their declarations and use implicit declarations (`patsubst %.cpp,%.o`). ChatGPT directly outputs variable assignments with full dependency lists (lines 3–7), which is a fix for explicit declarations and does not apply to implicit declarations. In contrast, Deepseek uses dynamic dependency tracking files for error fixing (*e.g.*, Deepseek fix, lines 8–9). MDfixer uses dif-


```
Errors:
Modules/expat/xmltok.o:  Modules/expat/xmltok_impl.c,
Modules/expat/xmltok_ns.c

Fixes from MDfixer
1.  $(DEPS_Modules_expat_xmltok) = Modules/expat/*.c
2.  Modules/expat/xmltok.o: $(DEPS_Modules_expat_xmltok)
    $(srcdir)/Modules/expat/xmltok.c $(LIBEXPAT_HEADERS)
    $(PYTHON_HEADERS)

Merged Fixes
3.  LIBEXPAT_HEADERS = + Modules/expat/xmltok_impl.c
    Modules/expat/xmltok_ns.c
4.  Modules/expat/xmltok.o: $(srcdir)/Modules/expat/xmltok.c
    $(LIBEXPAT_HEADERS) $(PYTHON_HEADERS)
```

Fig. 10: Merged Fixes in *CPython* Project. Modification Is Marked in Green.

TABLE II: Time Consumption and Results of Mdfixer Fixes and Fixes With Different LLMs (Success (✓), Failure (✗))

Project	Conversion	DS	GPT	Only DS	Only GPT	MDfixer (DS)	MDfixer (GPT)	Only DS	Only GPT
CPython	0.046	-	-	536.709	359.604	✓	✓	✔	✗
kleaver	0.007	-	-	10.977	418.676	✓	✓	✗	✗
clib	0.008	-	-	13.507	359.323	✓	✓	✓	✓
zlib	0.005	-	-	3.556	71.435	✓	✓	✗	✓
Generic-C	0.009	-	-	6.589	77.873	✓	✓	✓	✓
fzy	0.009	-	-	7.920	100.822	✓	✓	✓	✗
tcc	0.009	-	-	31.307	311.783	✓	✓	✓	✓
CacheSimulator	0.010	-	-	3.192	25.902	✓	✓	✓	✓
iodine	0.008	264.850	16.097	28.090	266.940	✓	✗	✓	✗
PacVim	0.008	90.406	11.397	4.098	82.380	✓	✗	✓	✗
sds	62.999	-	-	1.225	16.409	✓	✓	✓	✓
mon	0.004	62.996	3.800	3.300	54.134	✓	✓	✓	✓
genann	0.005	63.708	3.237	3.320	71.026	✓	✗	✗	✗
pixiewps	0.009	343.746	18.402	20.602	335.626	✓	✗	✓	✗
nyancat	0.005	282.560	4.228	6.617	245.633	✓	✗	✓	✗
coroutine	0.003	56.652	1.402	1.657	158.668	✓	✓	✓	✓
libsass	0.014	297.017	30.570	27.420	313.461	✓	✗	✓	✗
jsmn	0.003	-	-	4.554	39.488	✓	✓	✗	✗
pwnat	0.004	-	-	5.390	205.880	✓	✓	✗	✓
fastText	0.005	-	-	15.318	97.678	✓	✓	✗	✓
libco	0.008	160.278	18.501	13.127	82.783	✓	✗	✓	✗
cppcheck	62.999	-	-	387.426	282.749	✓	✓	✓	✓
ck	0.006	-	-	9.884	162.382	✓	✓	✗	✓
gravity	0.008	171.850	15.081	10.629	55.799	✓	✓	✓	✓
mpc	0.008	175.448	11.648	10.722	254.347	✓	✓	✓	✓
coturn	0.006	-	-	130.916	362.111	✓	✓	✓	✓
n2n	0.008	223.772	6.551	195.055	47.655	✓	✗	✓	✗
jemalloc	0.017	502.917	446.34	714.251	382.561	✓	✓	✓	✓
libhv	0.012	276.124	17.533	252.723	22.048	✓	✓	✓	✓
fio	0.015	585.519	242.756	289.573	585.317	✓	✗	✗	✗
sysstat	0.015	-	-	674.877	309.472	✓	✓	✓	✓
unbound	0.025	549.954	788.288	900.809	179.519	✓	✓	✗	✗
liburing	0.007	49.171	198.511	171.735	57.181	✓	✗	✓	✗
rsync	0.018	380.282	106.931	348.818	32.778	✓	✓	✓	✓
tcpdump	0.021	482.228	116.749	358.892	214.092	✓	✗	✓	✗
Cost	-	-	-	-	-	0.85$	1.38$	1.41$	2.49$

-: No fixes with LLMs. ✔: The original Makefile exceeds the output limitations of the LLM, and the output limitations can be resolved by using the Makefile processed by MDfixer.

```
OBJS:= $(patsubst %.cpp,%.o,$(wildcard src/*.cpp))
$(TARGET): $(OBJS)
# GPT Fixes
src/globals.o: src/globals.h
src/game.o: src/ghost1.h src/helperFns.h src/avatar.h src/globals.h
src/avatar.o: src/helperFns.h src/avatar.h src/globals.h
src/ghost1.o: src/ghost1.h src/helperFns.h src/avatar.h src/globals.h
src/helperFns.o: src/helperFns.h src/globals.h
# Deepseek Fixes
DEPFILES: = $(OBJS: .o=.d)
-include $(DEPFILES)
```

Fig. 11: Use Different LLMs to Fix PacVim

```
# GPT Fixes
1. zutil.o: $(SRCDIR)zutil.c zlib/gzguts.h
2. zutil.lo: $(SRCDIR)zutil.c zlib/gzguts.h
# Deepseek Fixes
3. adler32.o zutil.o: $(SRCDIR)zutil.h $(SRCDIR)zlib.h zconf.h
   $(SRCDIR)gzguts.h
4. adler32.lo zutil.lo: $(SRCDIR)zutil.h $(SRCDIR)zlib.h zconf.h
   $(SRCDIR)gzguts.h
```

Fig. 12: Only Use LLMs to Fix Zlib

ferent LLMs costing 0.85$ (Deepseek) and 1.38$ (ChatGPT), respectively.

b) Only Use LLMs: The results of fixing MDs using only LLMs are that Deepseek succeeded in fixing 73% (22/30) projects, and ChatGPT succeeded in fixing 53% (16/30) projects. Deepseek fails because it does not add the dependencies to the expected locations correctly. As shown in Fig 12, the correct fix would have been to insert 'zlib/gzguts.h' directly into the declaration of 'zutil.o' (lines 1–2), but Deepseek fixed it by adding extra declarations at the end (lines 3–4). MDfixer can correctly locate and fix errors through dependency graphs and style-aware fixing. Overall, LLMs find it difficult to apply different fixing strategies to explicit declarations and implicit declarations. They tend to use a single fixing pattern, making it difficult to solve all problems. The cost of the fix using only LLMs is 1.41$ (Deepseek) and 2.49$ (ChatGPT).

Answer to Q3: *MDfixer successfully fixed 35 projects using Deepseek and 24 projects using ChatGPT. It successfully fixed 22 projects using only Deepseek and 16 projects using only ChatGPT. This demonstrates the effectiveness of MDfixer.*

E. Artifact Consistency (Q4)

To answer this question, in 35 projects, we compared whether the fixes applied by MDfixer can unintentionally change the artifacts. We instructed EChecker to output the actual dependency graphs before and after the fix, and then calculated the similarity of the two actual dependency graphs.

a) Results: The two graphs of all projects is identical. This means that the fixes introduced by MDfixer in the Makefiles did not cause any changes in the building process. Although changes to the Makefile can easily lead to changes in the build process, which in turn may lead to different artifacts being built, MDfixer's fix for MDs does not impact the original build process of the project. This is because the MDs themselves are dependencies that the user missed declaring in the Makefile, and these dependencies themselves are used in the software build. Therefore, MDfixer does not change the build process even if it modifies Makefiles to add these undeclared dependencies.

Answer to Q4: *In subjects, after MDfixer fixes, the two actual dependency graphs are all identical, which demonstrates that the MDfixer fixes do not change the artifacts.*

V. Discussion

This section further discusses the issues relevant to MDfixer and this study.

a) MDs fixes from MDfixer and project maintainers: The fixes by MDfixer can be as effective as those by developers. Project maintainers commonly fix MDs based on the declared structure, as does MDfixer. In Fig. 8, the project maintainers used a macro to fix MDs without fixing the missing header files. MDfixer can declare a new macro for these header files and use it to fix the missing header files. Another example comes from the *cppcheck* project, whose maintainers fixed MDs in commit afe05d0. MDfixer, like *cppcheck*'s maintainers, generates the fix by adding each dependency in turn to the target's declaration. The advantage of MDfixer is that it does not rely on the developer's experience with Makefiles, it can automatically and efficiently fix MDs in the Makefile. Even for large-scale projects like *CPython*, MDfixer can complete the fixing process at an extremely fast speed (0.046s).

b) Build failures and different artifacts: As shown in Section IV-E, although MDfixer's fixes to MDs modify the build scripts, it does not cause build failures or different artifacts. Since MDfixer only fixes errors from the source files, build failures due to intermediate artifacts declared in Makefiles are avoided. In addition, MDs themselves are the dependencies that are used during the actual build, hence, declaring these dependencies in Makefiles does not affect the build, resulting in different build artifacts.

c) Adapt to other advanced build tools: The core innovation of MDfixer is to generate patches that are consistent with the project's unique declaration style, which can be adapted to other high-level build tools (*e.g.*, CMake and Autotools). Its current implementation delivers robust performance within the Makefile ecosystem, where complex and varied stylistic patterns make automatic fixes particularly challenging. Notably, our approach and implementation directly support the Autotools build system. Projects using CMake or Autotools rely on specifications (*e.g.*, `CMakeLists.txt` or `Makefile.in`) to generate Makefiles. MDfixer can directly address MDs in `Makefile.in` files. `Makefile.in` is a template file in the Autotools toolchain, containing variables and placeholders that get configured (via `./configure`) to generate system-specific Makefiles. `Makefile.in` contains build declarations (targets, dependencies, and compilation rules) with variable placeholders, which are substituted during configuration to generate system-specific Makefiles. MDfixer can fix these build declarations with placeholders. As demonstrated in the ck and libhv projects, fixes generated by MDfixer for such files have been successfully merged by maintainers.

For errors originating in `CMakeLists.txt` (rather than the generated Makefiles), the idea behind MDfixer would also apply to CMake and related build script generators. To this end, it would be necessary to extend MDfixer's error localization and patch generation mechanisms to parse the syntax and dependency declaration patterns of CMake and other related build script generators. This includes adapting the existing declaration graph model to accommodate CMake-specific constructs (*e.g.*, target_link_libraries, add_dependencies) and refining style recognition metrics to align with CMake's idiomatic declaration practices.

d) Complexity of subjects: Among the 35 projects, the maximum number of lines in the Makefile is 3,266 (CPython), and the minimum is 8 (Sds). MDfixer modified these Makefiles by a maximum of 278 lines, a minimum of 1 line, an average of 25 lines, and a median of 10 lines. Overall, the average fix is as large as 25 lines, which shows that these projects are not simple cases.

e) Threats to validity: The main threat to the internal validity of our evaluation is fluctuations in the performance of experimental equipment. To minimize randomness, we stopped all other user processes on the system and conducted five experiments for each project. The results are the median time spent on each experiment. In addition, we calculated the standard deviation of the total time spent in five experiments for all projects (*cf.* supplementary materials).

The main threat to the external validity of our evaluations lies in the selection of experimental projects. One potential threat might be to what extent the results could be generalized to other projects that we did not consider. To mitigate this risk, we selected a large set of projects. In addition, these projects have been used in other studies [12], [10], [11]. Since these projects have been thoroughly evaluated and widely used in other related studies, they can be the representatives of well-known projects that include build dependency errors.

One possible threat to the construct validity of our study is associated with the similarity of actual dependencies (*cf.* Section IV-E) to compare the similarity of artifacts. The actual dependencies are not a complete representation of the artifacts. However, artifacts are built from actual dependencies. Therefore, we consider it reasonable to calculate the similarity of artifacts using actual dependencies.

Another threat to the construct validity might be taking EChecker's detection results as ground truth. This ground truth may not be entirely true. However, EChecker, as the state-of-the-art dependency error detecting method, offers high effectiveness on dependency error detection [12], [23]. In an evaluation on 12 projects (240 total commits, 20 per project), EChecker achieved an F1 score 0.18 higher than Buildfs [16], with no explicit mention of false positives. In addition, we obtained the results of MDfixer to fix MDs by comparing the detection results of EChecker before and after fixes, which reduced the impact of EChecker's false positives on the experiment results. We consider the use of EChecker's detection results as ground truth is reasonable.

VI. Related Work

a) Program fixing: Practitioners usually make mistakes when writing code, and many program repair techniques [25], [26], [27], [28], [29] can help practitioners fix them. Hassan et al. [19] utilized build script history patches to fix software build crashes. Lou et al. [20] proposed a search-based technique that generates potential patches by searching for the present project and external resources rather than historical fixing information. Tarlow et al. [30] proposed a deep learning-based approach for fixing build errors. Joshi et al. [31] proposed a code repair engine with multilingual support that uses a large language model trained on code (such as Codex [32], an AI model capable of generating source code from plain English prompts). Oh et al. [33] proposed a way to fix type errors in Python projects. They identified error locations and patched candidates efficiently by inferring the correct and incorrect types of program variables and exploiting the differences between them. Kang et al. [34] proposed a way to fix build errors in Gradle [5], where they generated eigenvectors from build error information for training two classification models for causes and solutions.

Existing methods do not take into account how to fix MDs. These build script fix methods either utilize historical repair information or use build error logs, but few of them are applicable when fixing MDs. Since MDs do not have historical fixing information or build error logs. We propose a new approach to automatically fix MDs.

b) Build dependency errors detection: Common build dependency errors are *missing dependencies* (MDs) and *redundant dependencies* (RDs). MDs may erroneously prevent the build targets from rebuilding in incremental builds. RDs cause a decrease in build efficiency. Xia et al. [35] and Zhao et al. [36] used MAKAO [37] to parse build scripts and then analyzed the source code to predict missing dependencies. Licker et al. [10] proposed a method to detect build dependency errors by triggering a large number of incremental builds. Their method triggers incremental builds by modifying the test files (timestamp) to observe if the desired files were rebuilt. Bezemer et al. [15] leveraged parsing build scripts and monitoring actual builds to detect build dependency errors. Sotiropoulos et al. [16] and Fan et al. [11] treated each build target as a task, detecting dependency errors based on build targets. Wu et al. [13] proposed removing the body of the function in each source code file to perform the build, which they called a virtual build, to accelerate build dependency error detection. Lyu et al. [12] proposed a method (called EChecker) to update the actual dependency graph in incremental builds for efficient and effective detection of build dependency errors.

c) Build maintenance: Tamrawi et al. [38] proposed a symbolic evaluation algorithm that processes Makefile and generates a Symbolic Dependency Graph (SDG), which expresses the dependencies between the build rules and the files through the build commands. It can detect code smells in the build scripts and help with renaming. Al-Kofahi et al. [17], [18] proposed a method to locate bugs in the building code that cause failures in the runtime build to help practitioners parse build scripts. Gallaba et al. [39] proposed a way to decouple the acceleration of the build from the underlying build tools, speeding up the build by inferring dependencies between build steps. Mukherjee et al. [40] used historical build logs and current build logs to identify the presence of dependency issues and fix the problem of non-repeatable builds of Python projects. Lyu et al. [41] proposed a method called BUDDI to accelerate the building of multiple software configurations.

d) Build reproducibility: Verifying build reproducibility can uncover some critical bugs. Ren et al. [42] searched for problematic files that led to an unreproducible build for the localization task. They also identified the root cause of unreproducible builds by tracing the build process across different environments [21] and later searched for similar patches based on historical knowledge to fix an unreproducible build [22].

VII. Conclusion

We present MDfixer, the first automatic solution for detecting and fixing missing dependencies (MDs) in Makefiles. An evaluation on 35 projects demonstrates complete MDs resolution with 75% (2,099/2,786) upstream-accepted fixes, achieved in 3.31 minutes average (62.999s median) per fix. While leveraging EChecker's diagnostics, MDfixer's adaptable design enables integration with tools like Mkcheck [10] and Buildfs [15] through simple configuration adjustments for error report parsing. MDfixer demonstrates significant potential to reduce Makefile dependency management overhead.

Acknowledgment

This work is supported by the Key Research and Development Program of Jiangsu Province (No.BE2021002-2), the National Natural Science Foundation of China (No.62202219, No.62302210), and the Innovation Projects and Overseas Open Projects of State Key Laboratory for Novel Software Technology (Nanjing University) (ZZKT2025A12, ZZKT2025B18, ZZKT2025B20, ZZKT2025B22, KFKT2025A17, KFKT2025A19, KFKT2025A20, KFKT2024A02, KFKT2024A13, KFKT2024A14, KFKT2023A09, KFKT2023A10). This work was also supported by a Ministry of Education (MOE) Academic Research Fund (AcRF) Tier 1 grant.

Supplementary materials and artifacts of MDfixer are available at https://github.com/Meiye-lj/MDfixer.git.

References

[1] (2023) Gnu make manual. [Online]. Available: https://www.gnu.org/software/make/manual/make.html

[2] (2023) Scons: an open source software construction tool. [Online]. Available: www.scons.org

[3] (2023) Apache ant manual. [Online]. Available: https://ant.apache.org/manual/

[4] (2023) Maven: A software project management and comprehension tool. [Online]. Available: https://maven.apache.org/

[5] (2023) Gradle build tool. [Online]. Available: https://gradle.org/

[6] (2023) An introduction to the autotools. [Online]. Available: https://www.gnu.org/software/automake/manual/automake.html

[7] K. Martin, *Mastering CMake : A Cross-Platform Build System*, 2010.

[8] H. Seo, C. Sadowski, S. G. Elbaum, E. Aftandilian, and R. W. Bowdidge, "Programmers' build errors: a case study (at google)," in *Proceedings of the 36th International Conference on Software Engineering, Hyderabad, India - May 31 - June 07, 2014*, P. Jalote, L. C. Briand, and A. van der Hoek, Eds. ACM, 2014, pp. 724–734. [Online]. Available: https://doi.org/10.1145/2568225.2568255

[9] S. McIntosh, B. Adams, T. H. D. Nguyen, Y. Kamei, and A. E. Hassan, "An empirical study of build maintenance effort," in *Proceedings of the 33rd International Conference on Software Engineering, Waikiki, Honolulu , HI, USA, May 21-28, 2011*, R. N. Taylor, H. C. Gall, and N. Medvidovic, Eds. ACM, 2011, pp. 141–150. [Online]. Available: https://doi.org/10.1145/1985793.1985813

[10] N. Licker and A. Rice, "Detecting incorrect build rules," in *Proceedings of the 41st International Conference on Software Engineering, 2019, Montreal, QC, Canada, May 25-31, 2019*, J. M. Atlee, T. Bultan, and J. Whittle, Eds. IEEE / ACM, 2019, pp. 1234–1244. [Online]. Available: https://doi.org/10.1109/ICSE.2019.00125

[11] G. Fan, C. Wang, R. Wu, X. Xiao, Q. Shi, and C. Zhang, "Escaping dependency hell: finding build dependency errors with the unified dependency graph," in *Proceedings of 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, USA, July 18-22, 2020*. ACM, 2020, pp. 463–474. [Online]. Available: https://doi.org/10.1145/3395363.3397388

[12] J. Lyu, S. Li, H. Zhang, Y. Zhang, G. Rong, and M. Rigger, "Detecting build dependency errors in incremental builds," pp. 1–12, 2024. [Online]. Available: https://doi.org/10.1145/3650212.3652105

[13] R. Wu, M. Chen, C. Wang, G. Fan, J. Qiu, and C. Zhang, "Accelerating build dependency error detection via virtual build," in *Proceedings of the 37th ACM International Conference on Automated Software Engineering, ASE 2022, Rochester, MI, USA, October 10-14, 2022*. ACM, 2022, pp. 5:1–5:12. [Online]. Available: https://doi.org/10.1145/3551349.3556930

[14] (2023) Actual and declared dependencies. [Online]. Available: https://docs.bazel.build/versions/main/build-ref.html\#actual_and_declared_dependencies

[15] C. Bezemer, S. McIntosh, B. Adams, D. M. Germán, and A. E. Hassan, "An empirical study of unspecified dependencies in make-based build systems," *Empir. Softw. Eng.*, vol. 22, no. 6, pp. 3117–3148, 2017. [Online]. Available: https://doi.org/10.1007/s10664-017-9510-8

[16] T. Sotiropoulos, S. Chaliasos, D. Mitropoulos, and D. Spinellis, "A model for detecting faults in build specifications," *Proc. ACM Program. Lang.*, vol. 4, no. OOPSLA, pp. 144:1–144:30, 2020. [Online]. Available: https://doi.org/10.1145/3428212

[17] J. Al-Kofahi, H. V. Nguyen, and T. N. Nguyen, "Fault localization for make-based build crashes," in *Proceedings of the 2014 IEEE International Conference on Software Maintenance and Evolution*, 2014, pp. 526–530. [Online]. Available: https://doi.org/10.1109/ICSME.2014.87

[18] J. M. Al-Kofahi, H. V. Nguyen, and T. N. Nguyen, "Fault localization for build code errors in makefiles," in *Proceedings of the 36th International Conference on Software Engineering, ICSE '14, Companion Proceedings, Hyderabad, India, May 31 - June 07, 2014*, P. Jalote, L. C. Briand, and A. van der Hoek, Eds. ACM, 2014, pp. 600–601. [Online]. Available: https://doi.org/10.1109/ICSME.2014.87

[19] F. Hassan and X. Wang, "Hirebuild: an automatic approach to history-driven repair of build scripts," in *Proceedings of the 40th International Conference on Software Engineering, 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, M. Chaudron, I. Crnkovic, M. Chechik, and M. Harman, Eds. ACM, 2018, pp. 1078–1089. [Online]. Available: https://doi.org/10.1145/3180155.3180181

[20] Y. Lou, J. Chen, L. Zhang, D. Hao, and L. Zhang, "History-driven build failure fixing: how far are we?" in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, 2019, Beijing, China, July 15-19, 2019*, D. Zhang and A. Møller, Eds. ACM, 2019, pp. 43–54. [Online]. Available: https://doi.org/10.1145/3293882.3330578

[21] Z. Ren, C. Liu, X. Xiao, H. Jiang, and T. Xie, "Root cause localization for unreproducible builds via causality analysis over system call tracing," in *Proceedings of the 34th IEEE International Conference on Automated Software Engineering, ASE 2019, San Diego, CA, USA, November 11-15, 2019*. IEEE, 2019, pp. 527–538. [Online]. Available: https://doi.org/10.1109/ASE.2019.00056

[22] Z. Ren, S. Sun, J. Xuan, X. Li, Z. Zhou, and H. Jiang, "Automated patching for unreproducible builds," in *Proceedings of the 44th ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*. ACM, 2022, pp. 200–211. [Online]. Available: https://doi.org/10.1145/3510003.3510102

[23] J. Lyu, S. Li, B. Liu, H. Zhang, G. Rong, C. Zhong, and X. Liu, "Detecting build dependency errors by dynamic analysis of build execution against declaration," *IEEE Trans. Software Eng.*, vol. 51, no. 6, pp. 1745–1761, 2025. [Online]. Available: https://doi.org/10.1109/TSE.2025.3566225

[24] (2025) Blind link of cpython fixes. [Online]. Available: https://github.com/python/cpython/issues/1195xx

[25] B. Berabi, J. He, V. Raychev, and M. T. Vechev, "Tfix: Learning to fix coding errors with a text-to-text transformer," in *Proceedings of the 38th International Conference on Machine Learning, ICML 2021, 18-24 July 2021, Virtual Event*, ser. Proceedings of Machine Learning Research, M. Meila and T. Zhang, Eds., vol. 139. PMLR, 2021, pp. 780–791. [Online]. Available: http://proceedings.mlr.press/v139/berabi21a.html

[26] M. Yasunaga and P. Liang, "Graph-based, self-supervised program repair from diagnostic feedback," in *Proceedings of the 37th International Conference on Machine Learning, ICML 2020, 13-18 July 2020, Virtual Event*, ser. Proceedings of Machine Learning Research, vol. 119. PMLR, 2020, pp. 10 799–10 808. [Online]. Available: http://proceedings.mlr.press/v119/yasunaga20a.html

[27] Y. Michihiro and L. Percy, "Break-it-fix-it: Unsupervised learning for program repair," in *Proceedings of the 38th International Conference on Machine Learning, ICML 2021, 18-24 July 2021, Virtual Event*, vol. 139. PMLR, 2021, pp. 11 941–11 952. [Online]. Available: http://proceedings.mlr.press/v139/yasunaga21a.html

[28] C. Geethal, M. Böhme, and V. Pham, "Human-in-the-loop automatic program repair," *IEEE Trans. Software Eng.*, vol. 49, no. 10, pp. 4526–4549, 2023. [Online]. Available: https://doi.org/10.1109/TSE.2023.3305052

[29] K. Liu, J. Zhang, L. Li, A. Koyuncu, D. Kim, C. Ge, Z. Liu, J. Klein, and T. F. Bissyandé, "Reliable fix patterns inferred from static checkers for automated program repair," *ACM Trans. Softw. Eng. Methodol.*, vol. 32, no. 4, pp. 96:1–96:38, 2023. [Online]. Available: https://doi.org/10.1145/3579637

[30] D. Tarlow, S. Moitra, A. Rice, Z. Chen, P. Manzagol, C. Sutton, and E. Aftandilian, "Learning to fix build errors with graph2diff neural networks," in *Proceedings of the 42nd International Conference on Software Engineering, Workshops, Seoul, Republic of Korea, 27 June - 19 July, 2020*. ACM, 2020, pp. 19–20. [Online]. Available: https://doi.org/10.1145/3387940.3392181

[31] H. Joshi, J. Cambronero Sanchez, S. Gulwani, V. Le, G. Verbruggen, and I. Radiček, in *Repair Is Nearly Generation: Multilingual Program Repair with LLMs*, vol. 37, no. 4, Jun. 2023, pp. 5131–5140. [Online]. Available: https://ojs.aaai.org/index.php/AAAI/article/view/25642

[32] (2025) Openai codex. [Online]. Available: https://openai.com/blog/openai-codex

[33] W. Oh and H. Oh, "Pyter: effective program repair for python type errors," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2022, Singapore, Singapore, November 14-18, 2022*, A. Roychoudhury, C. Cadar, and M. Kim, Eds. ACM, 2022, pp. 922–934. [Online]. Available: https://doi.org/10.1145/3540250.3549130

[34] M. Kang, T. Kim, S. Kim, and D. Ryu, "Gradle-autofix: An automatic resolution generator for gradle build error," *Int. J. Softw. Eng. Knowl. Eng.*, vol. 32, no. 4, pp. 583–603, 2022. [Online]. Available: https://doi.org/10.1142/S0218194022500218

[35] X. Xia, D. Lo, X. Wang, and B. Zhou, "Build system analysis with link prediction," in *Symposium on Applied Computing, SAC 2014, Gyeongju, Republic of Korea - March 24 - 28, 2014*, Y. Cho, S. Y. Shin, S. Kim, C. Hung, and J. Hong, Eds. ACM, 2014, pp. 1184–1186. [Online]. Available: https://doi.org/10.1145/2554850.2555134

[36] B. Zhou, X. Xia, D. Lo, and X. Wang, "Build predictor: More accurate missed dependency prediction in build configuration files," in *Proceedings of the IEEE 38th Annual Computer Software and Applications Conference, COMPSAC 2014, Vasteras, Sweden, July 21-25, 2014*. IEEE Computer Society, 2014, pp. 53–58. [Online]. Available: https://doi.org/10.1109/COMPSAC.2014.12

[37] B. Adams, H. Tromp, K. D. Schutter, and W. D. Meuter, "Design recovery and maintenance of build systems," in *Proceedings of 23rd IEEE International Conference on Software Maintenance (ICSM 2007), October 2-5, 2007, Paris, France*. IEEE Computer Society, 2007, pp. 114–123. [Online]. Available: https://doi.org/10.1109/ICSM.2007.4362624

[38] A. Tamrawi, H. A. Nguyen, H. V. Nguyen, and T. N. Nguyen, "Build code analysis with symbolic evaluation," in *Proceedings of the 34th International Conference on Software Engineering, 2012, June 2-9, 2012, Zurich, Switzerland*. IEEE Computer Society, 2012, pp. 650–660. [Online]. Available: https://doi.org/10.1109/ICSE.2012.6227152

[39] K. Gallaba, J. Ewart, Y. Junqueira, and S. McIntosh, "Accelerating continuous integration by caching environments and inferring dependencies," *IEEE Trans. Software Eng.*, vol. 48, no. 6, pp. 2040–2052, 2022. [Online]. Available: https://doi.org/10.1109/TSE.2020.3048335

[40] S. Mukherjee, A. Almanza, and C. Rubio-González, "Fixing dependency errors for python build reproducibility," in *Proceedings of the ISSTA '21: 30th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, Denmark, July 11-17, 2021*, C. Cadar and X. Zhang, Eds. ACM, 2021, pp. 439–451. [Online]. Available: https://doi.org/10.1145/3460319.3464797

[41] J. Lyu, S. Li, H. Zhang, L. Yang, B. Liu, and M. Rigger, "Towards efficient build ordering for incremental builds with multiple configurations," *Proc. ACM Softw. Eng.*, vol. 1, no. FSE, pp. 1494–1517, 2024. [Online]. Available: https://doi.org/10.1145/3660774

[42] Z. Ren, H. Jiang, J. Xuan, and Z. Yang, "Automated localization for unreproducible builds," in *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, M. Chaudron, I. Crnkovic, M. Chechik, and M. Harman, Eds. ACM, 2018, pp. 71–81. [Online]. Available: https://doi.org/10.1145/3180155.3180224