# EvidenT: An Evidence-Preserving Framework for Iterative System-Level Package Repair

ANONYMOUS AUTHOR(S)

Frequent toolchain updates and the expanding diversity of instruction set architectures (ISAs) have made large-scale system-level software package repair a critical task. Diagnosing and repairing build failures remains challenging due to heterogeneous failure evidence, complex dependency constraints, and architecture-specific build conventions. While recent LLM-based repair methods have shown promise for project-level source code fixes, they struggle with system-level repair where failures involve multi-language artifacts (e.g., build recipes, scripts, and source archives) and require iterative validation through external build services. In this paper, we first conduct a systematic empirical study of real-world system-level build failures. Our findings reveal that 72% of failures stem from dependency and environment misconfigurations rather than isolated code defects, suggesting that effective repair must prioritize packaging logic and iterative feedback. Motivated by these insights, we propose EvidenT, an evidence-preserving repair framework that decouples iteration-aware evidence management from tool execution. EvidenT comprises (1) an external Build Service for reproducible build execution and feedback; (2) an Evidence-Preserving Repair Controller that performs cross-modal fusion of repair history, knowledge context, and build artifacts; and (3) an automated Repair Orchestrator that executes a suite of modular tools for failure localization and system-level repair actions within a closed-loop validation environment. We evaluate EvidenT on a benchmark of 219 real-world RISC-V package build failures. EvidenT successfully repairs 118 packages (53.88%), substantially outperforming state-of-the-art agentic baselines (20.55%) and direct LLM-based repair (1.83%). To demonstrate its architectural generality, we extend EvidenT to legacy ISAs by updating only ISA-specific knowledge context. In preliminary experiments, it achieves success rates of 41.77% on aarch64 and 46.99% on x86_64, showcasing its robustness across diverse hardware ecosystems.

Additional Key Words and Phrases: System-Level Package Repair, Build Failures, Tool Orchestrator, Evidence-Preserving, Iterative Repair

**ACM Reference Format:**
Anonymous Author(s). 2026. EvidenT: An Evidence-Preserving Framework for Iterative System-Level Package Repair. 1, 1 (April 2026), 21 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

## 1 Introduction

As software ecosystems continue to evolve, large-scale system-level package builds triggered by toolchain updates, system component upgrades, and architecture evolution have become routine. Despite extensive automation, build failures remain inevitable in practice due to complex dependency graphs, architecture-specific constraints, and fragmented *failure evidence* scattered across build stages. Once such failures occur, repair is often labor-intensive and costly, delaying software releases and incurring substantial maintenance effort. For example, Apple's transition from Intel x86 to ARM-based M1 processors required rebuilding thousands of packages, many of which encountered compatibility problems that took months to resolve [11].

As summarized in Table 1, existing approaches exhibit significant limitations when applied to system-level packages. Traditional approaches [9, 16, 20, 29, 32, 39, 45] primarily rely on handcrafted

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2026 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM XXXX-XXXX/2026/4-ART
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Table 1. Capability matrix of related approaches compared to EvidenT. "Category" indicates the methodology class, and checkboxes denote supported features essential for system-level package repair.

| Category | Method | Multi-language | Heterogeneous-artifact Pipeline | Automatic Repair | System-level Build |
|---|---|---|---|---|---|
| Traditional Approaches | HireBuild / History-Driven Fixing [16] | ✗ | ✗ | ✓ | ✗ |
| | Escaping Dependency Hell [9] | ✓ | ✗ | ✗ | ✗ |
| | My Fuzzers Won't Build [32] | ✓ | ✗ | ✗ | ✗ |
| | BuildSonic [45] | ✓ | ✗ | ✗ | ✗ |
| | Automatic Build Repair [29] | ✗ | ✗ | ✓ | ✗ |
| | Shipwright [20] | ✓ | ✗ | ✗ | ✗ |
| | Intelligent Automation [39] | ✗ | ✗ | ✓ | ✗ |
| Agent-based Approaches | RepairAgent [3] | ✗ | ✗ | ✓ | ✗ |
| | CXXCrafter [43] | ✗ | ✗ | ✓ | ✗ |
| | Agentless [42] | ✗ | ✗ | ✓ | ✗ |
| **Ours** | **EvidenT** | ✓ | ✓ | ✓ | ✓ |

rules, historical repair patterns, or specific heuristics tailored to isolated failure modes. They typically fail to generalize to real-world package ecosystems characterized by multi-language environments, heterogeneous-artifacts (e.g., packaging recipes, build scripts, and source archives), and the need for end-to-end build validation.

While Large Language Models (LLMs) have been explored as automated assistants for log analysis and software debugging [18, 19], our evaluation on 219 failed packages from the Open Build Service (OBS)[1] reveals a clear limitation: LLMs successfully repaired only 4 cases when directly analyzing build logs without external tool support. This suggests that LLMs, in isolation, cannot handle the complexity of system-level package repair. Recent agent-based extensions [3, 42, 43] improve automation through multi-step tool use. Nevertheless, they typically designed for source-centric or single-artifact environments and still lack explicit mechanisms to preserve and reuse failure evidence across iterations. Consequently, these agents frequently repeat ineffective actions and fail to converge under heterogeneous and architecture-specific conditions.

Our empirical study in Section 3 demonstrates that system-level package failures are highly diverse. Specifically, 56% of failures stem from dependency, compilation, or packaging issues, while the remaining 44% occur during testing and validation. Furthermore, among successful repairs, 72% primarily involve adjustments to build configurations, dependency, or environment settings, and 28% require modifications to source archives. These findings indicate that effective automated repair cannot rely solely on isolated log inspections or source-centric reasoning.

In practice, system-level package repair produces rich *failure evidence*, including build feedback, dependency constraints, package structure, and prior repair outcomes. However, such evidence is often fragmented, noisy, and iteration-dependent, making it difficult to exploit effectively in automated repair. Based on our empirical observations, we distill three fundamental challenges:

**Challenge 1: How to acquire actionable failure evidence.** Failure evidence is submerged within voluminous build logs, complex dependency resolution outputs, and heterogeneous package artifacts. Extracting precise repair guidance from such noisy data, while compensating for the agent's lack of architecture-specific experience through a knowledge context, remains a primary obstacle.

**Challenge 2: How to organize and fuse heterogeneous evidence.** Evidence signals span multiple modalities, such as semi-structured text (logs), graph structures (dependency constraints), and cached findings from multiple tools. Organizing these heterogeneous evidence streams into a unified and structured representation is essential for guiding artifact-level repair decisions.

[1] OBS is an open build system that supports package building: https://openbuildservice.org/

**Challenge 3: How to preserve and reuse iteration-dependent evidence.** System-level package repair is an iterative process of analysis, modification, and validation. Most existing methods treat each attempt in isolation, failing to preserve evidence across iterations. Without explicit evidence preservation, agents often repeat ineffective actions and explore redundant repair paths, leading to unstable repair loops.

These challenges motivate the design of a repair framework that can coordinate multiple tools and explicitly preserve failure evidence across repair iterations. To this end, we propose **EvidenT**, an evidence-preserving framework for iterative system-level package repair. EvidenT integrates analysis, repair, and validation into a unified workflow, in which repair outcomes are continuously verified through external build services such as OBS and Docker. To support scalable and decoupled tool orchestration, EvidenT adopts the Model Context Protocol (MCP) [21] as a standardized substrate for tool interaction.

Architecturally, EvidenT comprises three main components. To distill actionable failure evidence from noisy build signals and to complement missing expertise with auxiliary knowledge (Challenge 1), we design an *Analysis and Repair Orchestration* module that coordinates specialized tools for failure localization, artifact inspection, and system-level repair actions across packaging recipes, archives, and source code. To unify and structurally fuse heterogeneous evidence streams (Challenge 2), we introduce an *Evidence-Preserving Repair Controller* that maintains an iteration-aware evidence context and injects fused evidence into fixed prompt slots for disciplined repair decisions. To preserve an iteration-aware evidence (Challenge 3), we incorporate a *Build-based Validation and Feedback Loop* that executes patched packages in reproducible external build environments and feeds build outcomes back to the controller as new failure evidence, preventing redundant exploration and supporting stable convergence.

In summary, this paper establishes evidence preservation as a necessary principle for scalable iterative repair of system-level packages, and makes the following contributions:

- **The first empirical characterization of system-level repair.** We present the first large-scale empirical study of real-world system-level package build failures, analyzing 219 failed packages from OBS and further examining 100 representative cases in depth to characterize root causes, repair targets, and failure stages across heterogeneous artifacts.
- **Evidence-preserving repair framework.** We propose EvidenT, an evidence-preserving framework that coordinates heterogeneous analysis, repair, and build-validation tools in a unified iterative workflow, enabling automated repair beyond source-centric and single-artifact settings.
- **Superior performance and generalization.** EvidenT achieves repair success rates of 53.88% on RISC-V [8], 41.77% on aarch64, and 46.99% on x86_64 OBS package failures, outperforming LLM-only repair and representative agent-based baselines.
- **Open-source implementation and datasets.** We release our implementation and datasets through an anonymous repository to support reproducibility and future research[2].

## 2 Background & Related Work

### 2.1 Package Building

**Overview.** Transforming source code and build specifications (e.g., spec files) into installable binary packages is a core task in software maintenance and distribution. Traditional build tools (e.g., `rpmbuild`, `dpkg-buildpackage`) as well as integrated platforms such as the Open Build Service (OBS) and Koji [10] provide automation in dependency resolution, compilation, and packaging[1, 14, 28]. Container-based solutions (e.g., Docker) further improve reproducibility

[2] https://anonymous.4open.science/r/EvidenT-1638/README.md

148 by encapsulating the build environment [7, 30, 31, 34]. While containerized approaches mitigate
149 environment inconsistencies, they do not eliminate other sources of build failures arising from pack-
150 age configuration, dependencies, or code-level issues. Consequently, builds still fail frequently due
151 to a wide range of factors spanning dependencies, source code, and build specifications [16, 17, 35].
152 **Challenges in Large-Scale, Multi-Language Builds.** Large-scale system-level package build-
153 ing fundamentally differs from project-level builds in both scope and complexity. Packages often
154 combine multiple programming languages, rely on heterogeneous build systems, and must be
155 validated across diverse architectures. Different ecosystems (e.g., C/C++, Java, Python) rely on
156 distinct build systems and dependency management tools [12, 13, 24], which complicates auto-
157 mated repair beyond single-language scenarios studied in prior work [15, 41, 43]. Existing studies
158 typically address only one dimension of this complexity. Java-focused work shows that some
159 failures can be mitigated by modifying build scripts or configuration files [15]. Python-oriented
160 research emphasizes dependency conflict resolution within package managers such as pip and
161 conda [41]. For C/C++ projects, CXXCrafter [43] reports that only a small fraction of projects can be
162 automatically built without substantial human intervention. Other research has explored fast build
163 triage and the impact of build system evolution [4, 36], but these techniques focus on diagnosis
164 rather than automated repair. Overall, existing studies address individual aspects of the problem,
165 while comprehensive automated repair for large-scale, multi-language, system-level package builds
166 remains an open challenge.
167
168 ## 2.2 Agent and MCP

169 **LLM-Based Agents.** Large language models (LLMs) have achieved notable success in knowledge
170 storage [44], code generation [22], and semantic understanding [40]. LLM-based agents integrate
171 these models with external tools, planning mechanisms, and environment interaction, enabling
172 more autonomous execution [2, 23, 26, 38]. In software engineering, agent-based systems have been
173 applied to automated debugging, program repair, and build-related tasks [5, 6, 27]. Representative
174 examples include RepairAgent [3], CXXCrafter [43], and Agentless [42], which demonstrate that
175 LLM-driven agents can automate project-level repairs or build fixes within constrained benchmarks
176 or single-language settings. However, these agents often assume that failures can be localized
177 within a limited set of source artifacts or a single language ecosystem. They lack native support for
178 coordinated manipulation of heterogeneous artifacts such as spec files, source archives, and build
179 metadata, and do not explicitly preserve or reason over failure evidence across repair iterations.
180 Consequently, their applicability to system-level package repair remains limited.
181 **Model Context Protocol (MCP).** The Model Context Protocol (MCP) standardizes interactions
182 between LLMs and external tools by providing a unified interface for tool invocation and stateful
183 communication. MCP supports cross-model interoperability [25], modular and reusable tool inte-
184 gration, dynamic discovery of tools at runtime [37], and structured state exchange via JSON-RPC.
185 Importantly, MCP enables the preservation and reuse of contextual information such as long build
186 logs and intermediate tool outputs across interactions [25]. These properties make MCP a suitable
187 substrate for implementing iterative, evidence-aware workflows in complex tool-based settings.
188 Nevertheless, MCP itself is domain-agnostic and does not address build-specific challenges such as
189 failure localization across artifacts or architecture-aware reasoning. A dedicated repair framework
190 is still required to leverage MCP effectively in system-level package repair.
191
192 ## 3 Empirical Study

193 In this section, we conduct an empirical study to characterize real-world system-level package
194 build failures and to understand the fundamental challenges that hinder automated repair. Using
195 build failures on the RISC-V ISA as a representative setting, we analyze failure causes, repair
196

197 difficulty, and cross-artifact modification patterns to derive actionable guidance for the design of
198 an automated system-level repair framework. The research questions, dataset, and key findings are
199 structured as follows.
200 **Research Questions.** We formulate the following research questions to understand the root causes
201 of system-level build failures and the practical challenges that limit automated repair:

202 • RQ1 (Failure Causes). What are the primary root causes of build failures observed among these
203 packages?
204 • RQ2 (Repair Difficulty). Why is repairing failed packages challenging, and how do failed packages
205 differ from their nearest successful builds in terms of repair locations and required modifications
206 across artifacts?

207 **Dataset.** We collect the 100 most recent failed system-level packages on RISC-V from the Open
208 Build Service (OBS), together with their complete build logs and accessible source directories. These
209 packages represent a subset of the final evaluation dataset and are selected to separate empirical
210 characterization from framework evaluation.
211 To support RQ2, for each failed build we additionally retrieve the temporally nearest *successful*
212 build of the same package (by build date) and treat it as a control group for comparative analysis. To
213 ensure validity and reproducibility, we apply three inclusion criteria: (1) each failed build is followed
214 by at least one subsequent successful build, enabling controlled comparison between failure and
215 success states; (2) complete package artifacts and corresponding failure logs are accessible for
216 diagnosis (e.g., spec files, sources, and auxiliary packaging files); and (3) both the failure and success
217 builds can be reliably reproduced using the original source code, which eliminates inconsistencies
218 from environmental variables.
219 The selected packages cover diverse technical domains, encompassing development toolchains,
220 cloud-native and container management, network services, security and cryptography, graphics
221 and multimedia, system monitoring, data analysis and programming support, hardware adaptation,
222 Python ecosystem and libraries, and command-line tools. We analyze the source code language
223 distribution of the 100 system software packages. Python dominates the dataset (48%), followed
224 by C/C++ (17%) and Go (10%). Smaller portions are implemented in Rust (3%), Ruby (3%), and
225 Perl (2%), while 17% adopt mixed-language implementations. For instance, `apache2-mod_wsgi` [3]
226 combines extension modules in C with application logic in Python, reflecting the cross-language
227 characteristics of certain system-level software packages and explains why agents designed for
228 specific language environments cannot be directly transferred, posing a challenge to our work.
229
230 ### 3.1 Failure Causes from Build Logs (RQ1)

231 To answer RQ1, we manually analyze and classify the build failure logs of the 100 software packages.
232 Two software engineers independently conduct fine-grained log inspection and root-cause labeling
233 using domain knowledge of Linux packaging and build systems; disagreements are resolved through
234 discussion to produce a unified taxonomy. Ultimately, as shown in Table 2, four primary causes of
235 build failures are identified: *Dependency Failures*, *Compilation Failures*, *Test Failures*, and *Packaging*
236 *Failures*. Detailed descriptions of these categories are provided below.
237 **Dependency Failures (21/100).** Dependency failures arise when required external components
238 (e.g., libraries, tools, or modules) are missing, misconfigured, or incompatible with the build environ-
239 ment. We observe 21/100 cases in this category, including *Missing Dependencies* (15) and *Dependency*
240 *Conflicts* (6). Representative examples include missing `wlroots` in Meson projects, absent `cmake`
241 required by the Rust crate `aws-lc-sys`, missing Python modules such as `pkg_resources`, and ver-
242 sion incompatibilities such as PyO3 under Python 3.13 or `quic-go` under Go 1.21. Comparing with
243
244 [3] https://build.opensuse.org/package/show/openSUSE:Factory/apache2-mod_wsgi
245

Table 2. Classification of Build Failure Cases. "Category" refers to the expert-defined failure category, "Sub-category" is the further division based on the failure, and "Count" indicates the number of packages in each sub-category.

| Category | Subcategory | Description | Count |
|---|---|---|---|
| Dependency Failures | Missing Dependency | Missing versions of required components, libraries, and modules in the build environment. | 15 |
| | Dependency Conflict | Version mismatch or reliance on deprecated dependencies. | 6 |
| | **Sum** | | **21** |
| Compilation Failures | Code-Level Issues | Errors related to syntax, type definitions, and logical inconsistencies in the source code. | 11 |
| | Build Errors | Issues with build scripts, compiler configurations, and compilation processes. | 11 |
| | API & Function Issues | Calling deprecated, removed, or incompatible functions and APIs. | 5 |
| | **Sum** | | **27** |
| Test Failures | Code & Compatibility Issues | Outdated, incompatible, or incorrect test code. | 11 |
| | Assertion & Compliance Violations | Violation of test assertions or compliance checks. | 17 |
| | Test Execution Problems | Failures related to the execution of the test framework itself. | 16 |
| | **Sum** | | **44** |
| Packaging Failures | Packaging Configuration Errors | Incorrect settings or syntax in the packaging specification and configuration files. | 6 |
| | Installation/Verification Failures | Problems occurring during the actual installation or final verification stages. | 2 |
| | **Sum** | | **8** |
| **Total** | | | **100** |

successful builds suggests that fixes often require adjusting `BuildRequires`/runtime dependencies, exposing dependencies via pkg-config/CMake, upgrading incompatible versions, or providing missing tools/headers.

**Compilation Failures (27/100).** Compilation failures occur when compilers reject source code or build instructions. We observe 27/100 cases, including *Code-level Issues* (11; e.g., missing headers, incomplete types, invalid callback signatures), *Build Errors* (11; e.g., invalid flags, unsupported options, or toolchain constraints such as LTO back-end failures under specific RISC-V vector configurations), and *API & Function Issues* (5; e.g., deprecated OpenSSL 3.0 APIs promoted to errors under `-Werror`, or removed C APIs in Python 3.13). Typical fixes include modernizing API usage, adding headers/feature guards, correcting compiler flags, or aligning code with updated dependency versions.

**Test Failures (44/100).** Test failures occur during validation (`%check`) when test suites fail. We observe 44/100 cases, including *Code & Compatibility Issues* (e.g., deprecated unittest methods, removed modules such as `imp` in Python 3.12, type mismatches), *Assertion & Compliance Violations* (e.g., incorrect assertions such as `assertEquals`, mismatched AST/XML outputs, compliance warnings treated as errors), and *Test Execution Problems* (e.g., misconfigured discovery, missing CTest binaries, GUI-induced segmentation faults). Fixes typically require updating assertions/APIs, correcting discovery paths and environment variables, and provisioning test-only dependencies to stabilize harness execution.

**Packaging Failures (8/100).** Packaging failures originate from packaging metadata or scripts. We observe 8/100 cases, including *Packaging Configuration Errors* (6; e.g., macro/syntax issues, phase ordering, install path misconfigurations such as `libtool` refusing installs outside `/usr`)

295 Table 3. Types of Repair Observed in 100 Failed Packages. "Category" refers to expert-defined modification
296 classes and "Modification details" show the concrete operations required for package repair
297

| Category | Description | Modification details | Count |
|---|---|---|---|
| Spec Modification | Complex configuration adjustments, such as correcting build macro definitions or updating dependency declarations. | modified lines $\leq$ 50 | 45 |
| | | modified lines in 51–100 | 23 |
| | | modified lines > 100 | 4 |
| | | **Sum** | **72** |
| Packaging-Related Changes | Vendor archives, service/metadata files, auxiliary data. | Added files | 9 |
| | | Deleted files | 2 |
| | | Modified files | 17 |
| | | **Sum** | **28** |

306
307 and *Installation/Verification Failures* (2; e.g., file list mismatches such as “glob not found”, or cross-
308 subpackage file conflicts). These failures are commonly addressed by fixing spec macros/phase
309 ordering, repairing install paths, and adding missing `BuildRequires`.
310
311 **Finding 1:** Build failures span multiple stages: **56%** arise from dependency/compilation/packag-
312 ing issues, while **44%** are test-related. Each category corresponds to distinct failure evidence
313 and repair locations, making accurate root-cause analysis essential for automated repair.
314
315 **Implication.** Failure categories correspond to orthogonal repair actions and target artifacts. With-
316 out explicit *Root Cause Analysis (RCA)*, automated repair easily degenerates into trial-and-error
317 over irrelevant files and log fragments. This motivates an RCA tool suite with two core functions:
318 (1) extracting and prioritizing anomalous log segments that are most likely causal, and (2) localizing
319 candidate artifacts for modification (e.g., spec files, manifests, source/header files, and test scripts).
320
321 ### 3.2 Repair Locations and Modification Complexity (RQ2)

322 To answer RQ2, we compare each failed package against its nearest successful build. We design an
323 automated workflow that extracts, aligns, and cross-validates differences across four dimensions: (1)
324 directory structure, (2) source files, (3) specification (spec) files, and (4) packaging-related archives
325 and auxiliary files. Specifically, the analysis is performed at both the file and line levels, allowing us
326 to capture fine-grained code edits, file insertions/deletions, and complex specification adjustments.
327 Applying this workflow to 100 paired builds yields the modification distribution in Table 3, with
328 two dominant categories.
329 **Spec Modifications (72/100).** Spec files govern build phases and dependency declarations. Most
330 successful repairs require configuration-level adjustments such as macro/flag corrections, depen-
331 dency updates, and patch instruction changes. Quantitatively, most spec modifications are relatively
332 small in scale, with $\leq$ 50 modified lines (45 cases). However, a non-trivial fraction required medium-
333 scale edits of 51–100 lines (23 cases), and a smaller but significant set involved large-scale rewrites
334 exceeding 100 lines (4 cases). This prevalence indicates that automated repair must prioritize
335 reasoning over build specifications and dependency configurations, rather than focusing solely on
336 source code.
337 **Packaging-Related Changes (28/100).** Beyond spec file modifications, a notable subset of repairs
338 targeted the source archives or auxiliary files. These interventions generally fall into three categories:
339 (1) *File additions* (9 cases): new files are introduced to address missing components, provide necessary
340 datasets, or integrate third-party libraries absent from the original package; (2) *File deletions* (2 cases):
341 obsolete or conflicting files are removed to prevent incompatibilities, such as legacy helper scripts
342 or outdated binaries that disrupted the build process; and (3) *File modifications* (17 cases): existing
343

scripts, metadata, or auxiliary configurations are revised to satisfy system-level requirements. Typical adjustments included updating `.service` files, refining `systemd` settings, and correcting metadata inconsistencies.

> **Finding 2:** In **72%** of cases, successful repair primarily involves spec-level configuration and dependency/environment adjustments, while the remaining **28%** require changes to archives or auxiliary package artifacts beyond the spec file.

**Implication.** This distribution shows that effective automated repair must reason across heterogeneous artifacts, rather than treating build failures as isolated source-code defects. Spec-level modifications often require non-trivial reasoning over build environments, dependency specifications, and architecture-specific configurations, even when changes are relatively localized. In contrast, packaging-related repairs demand accurate comprehension of source archives and packaging conventions, where insufficient structural understanding can lead to incorrect or incomplete modifications.

## 4 EvidenT

In view of the challenges identified in Section 3, we design EvidenT, an evidence-preserving framework for automated system-level package repair. EvidenT targets build failures arising from heterogeneous artifacts, distributed failure signals, and iterative repair dependencies that commonly occur in large-scale package ecosystems.

### 4.1 Overview

Figure 1 illustrates the overall architecture of EvidenT. The framework separates evidence tracking from tool execution and comprises three components: (1) an external *Build Service* that provides reproducible build results and concrete failure logs, (2) an *Evidence-Preserving Repair Controller* that maintains iteration-aware repair context and constructs a dynamic prompt to guide repair actions, and (3) an MCP-based tool layer that executes *Analysis and Repair Orchestration* and *Validation and Feedback Loop*. Together, these components form an iterative loop that analyzes failures, applies targeted repairs, and validates outcomes through real builds.

### 4.2 Evidence-Preserving Repair Controller

The Evidence-Preserving Repair Controller maintains an iteration-aware evidence context, conditioning each repair iteration on the latest build feedback, prior edits, cached analysis results, and relevant domain knowledge. This design prevents repeated ineffective modifications and enables systematic, evidence-driven repair.

*4.2.1 Evidence Components.* EvidenT organizes the evidence context into four complementary components. Together, these four components answer: (1) what failed (Build Feedback), (2) what has been tried (Repair History), (3) what we know about the current package state (Cached Findings), and (4) what similar failures looked like before (Knowledge Context).

**Build Feedback.** To ensure iteration-aware failure analysis grounded in observable build behavior, EvidenT captures iteration-level failure evidence produced by the external build service. In EvidenT, each repair iteration is validated through a clean rebuild on the Open Build Service (OBS), yielding (i) the complete build log and (ii) a structured build outcome (e.g., succeeded, failed, or unresolvable). Only the most recent build feedback is retained. This design enforces a strict iteration boundary, ensuring that failure analysis is always grounded in the latest observed behavior and prevents

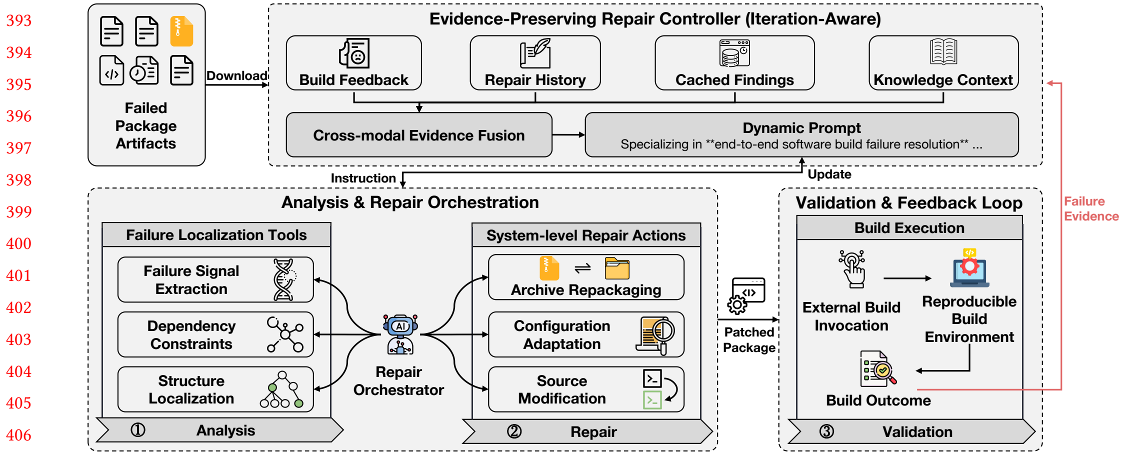



Fig. 1. The framework of EvidenT. The evidence-preserving repair controller maintains iteration-aware failure evidence, while the tool orchestrator exposes analysis, repair, and validation tools.

outdated logs from misleading later iterations. The build feedback serves as the primary diagnostic signal for subsequent failure localization and repair planning.

**Repair History.** To avoid repeated ineffective or contradictory repair actions across iterations, EvidenT explicitly maintains a repair history that records all artifact-level modifications applied in previous iterations. Each history entry captures the modified file path together with its updated content. Unsuccessful repair actions are preserved as negative evidence and explicitly exposed to the LLM in subsequent iterations. By conditioning repair decisions on accumulated modification history, EvidenT enables progressive refinement of repair strategies.

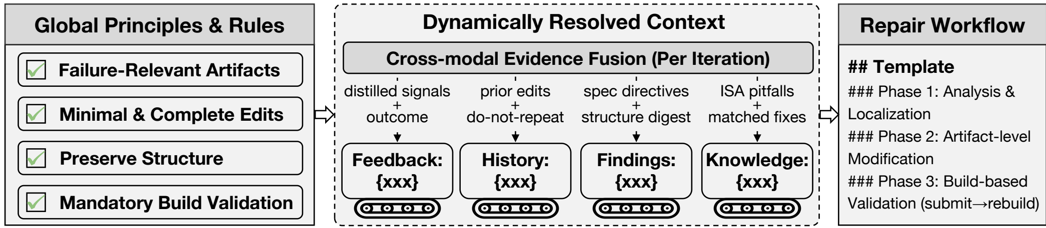
**Cached Findings.** To ensure the consistency of multi-step reasoning, EvidenT caches the outputs of deterministic analysis tools, such as directory structure summaries and parsed packaging recipes. These results remain invariant unless the underlying artifacts are modified within the same iteration. This mechanism minimizes redundant tool invocations and token overhead, thereby maintaining a stable context for reasoning. Cached entries are refreshed at the beginning of each iteration to reflect any artifact updates introduced by the previous repair attempt.

**Knowledge Context.** To provide auxiliary evidence for architecture-aware and ecosystem-specific reasoning, EvidenT maintains a knowledge context. It aggregates curated knowledge derived from historical issue discussions, pull-request reviews, and architecture-specific documentation related to system-level package failures. This knowledge is retrieved on demand and complements build-derived evidence by encoding recurring failure patterns, toolchain conventions, and ISA-specific pitfalls that are not explicitly exposed in build logs. By integrating such contextual evidence, EvidenT enables informed repair decisions beyond isolated source-level analysis.

*4.2.2 Cross-modal Evidence Fusion.* To transform heterogeneous repair signals into actionable instructions, EvidenT performs *cross-modal evidence fusion*. Rather than naively concatenating data, the system integrates build feedback, repair history, cached findings, and knowledge context into a unified evidence representation.

The fusion process enforces three principles. First, *temporal grounding* ensures strict iteration boundaries by retaining only the failure evidence produced by the most recent rebuild. Second, a *slot-based prompt structure* organizes the four evidence components into a fixed, non-overlapping schema. Instead of interleaving raw snippets, EvidenT (i) distills *Build Feedback* into a concise set

442 443 444 445 446 447 448 449




450 Fig. 2. Compact prompt schema of EvidenT. Per iteration, cross-modal evidence fusion organizes four evidence
451 components into fixed prompt slots, combined with global repair rules and a build-validated workflow.

452 453

454 of distilled failure signals and the current build outcome; (ii) represents *Repair History* as an ordered
455 list of prior edits alongside their validation results; (iii) normalizes *Cached Findings* into structured
456 key summaries with stable identifiers; and (iv) injects *Knowledge Context* only when it is relevant to
457 the current failure. This structured organization makes the prompt explicitly iteration-aware while
458 reducing ambiguity caused by redundant or inconsistent evidence sources. Third, *negative-evidence*
459 *constraints* encode unsuccessful edits as explicit "do-not-repeat" rules tied to file paths and change
460 types, which prunes redundant actions in subsequent iterations.
461
462 *4.2.3 Dynamic Prompt.* EvidenT constructs a dynamic prompt before each repair iteration to
463 translate iteration-aware evidence into more focused repair actions. The prompt serves as the main
464 control interface for the LLM agent that constrains how the LLM analyzes failures, applies artifact-
465 level edits, and performs build-based validation. Figure 2 summarizes this schema, combining global
466 repair rules with fixed evidence slots produced by cross-modal fusion and an explicit validation-
467 driven workflow.
468 **Global Principles and Procedural Rules.** The prompt encodes non-negotiable constraints that
469 govern repair behavior. These include evidence-guided prioritization of failure-relevant artifacts,
470 enforcing minimal and complete edits, preserving original file structure and formatting, and
471 requiring that all modifications be validated through the external build service. In particular, each
472 repair attempt must terminate with an explicit build submission, and unsuccessful validation
473 automatically triggers the next iteration. These rules prevent premature termination and ensure
474 that all reported repairs are grounded in real build outcomes.
475 **Dynamically Resolved Context.** At each iteration, the prompt instantiates four fused evidence
476 slots (Feedback, History, Findings, and Knowledge), making the latest failure evidence, prior edits,
477 and relevant contextual guidance explicit. This slot-based organization reduces ambiguity from
478 unstructured logs and supports history-aware reasoning across iterations.
479 **Structured Repair Workflow.** To further reduce ambiguity in long-horizon reasoning, the prompt
480 embeds a task-oriented workflow that decomposes repair into three ordered phases: failure analysis
481 and localization, targeted artifact-level modification, and build-based validation. Each phase is
482 illustrated with lightweight examples and tool usage patterns, making the intended repair logic
483 explicit. This design mitigates unstructured exploration and encourages systematic, reproducible
484 repair behavior across iterations.
485
486 ## 4.3 Analysis and Repair Orchestration
487 Under the guidance of the Evidence-Preserving Repair Controller, this module executes the analysis–
488 repair stages of each iteration. As illustrated in Figure 1, it comprises two functional groups: *Failure*
489 *Localization Tools* for the *Analysis* phase and *System-level Repair Actions* for the *Repair* phase.
490

491 *4.3.1 Failure Localization Tools.* To identify which artifacts are most likely responsible for the
492 failure, EvidenT provides three localization tools that summarize (1) the key error from the build
493 log, (2) relevant dependency/configuration constraints from the recipe, and (3) a lightweight view
494 of the package structure.
495 **Failure Signal Extraction.**
496 To transform voluminous and heterogeneous build logs into actionable repair guidance, EvidenT
497 employs a unified extraction tool that produces compact failure signals grounded in the latest build
498 feedback. The tool follows a two-stage process.
499 First, it performs *anomaly-focused log condensation* to isolate failure-relevant segments from
500 background noise. This stage purifies the raw logs by removing redundant entries and replacing
501 complex paths with concise placeholders. By cutting through this log noise, EvidenT prevents the
502 model from being overwhelmed and ensure it prioritizes salient failure evidence. The extractor fur-
503 ther applies phase-aware parsing to ensure that the distilled evidence remains strictly aligned with
504 the temporal order of build stages such as `configure` or `install`. Following this pre-processing,
505 the tool extracts contextual windows around diagnostic keywords (e.g., *undefined reference*, *fatal*
506 *error*) and normalizes them into abstract event templates using Drain [46]. To ensure high recall,
507 an LLM-based verifier classifies these candidate blocks to retain those containing explicit error
508 patterns, which, together with their build-stage metadata, constitute the distilled failure signals.
509 Second, it performs *retrieval-based contextualization* based on these distilled failure signals to
510 query the controller-maintained *Knowledge Context*. The tool leverages the extracted templates,
511 error keywords, and architecture hints within the signals to retrieve semantically related historical
512 failures and ISA-aware knowledge via TF–IDF indexing and cosine similarity. This retrieved context
513 complements the raw log evidence with recurring failure patterns and ecosystem conventions,
514 thereby providing the domain expertise required for complex system-level repair.
515 **Dependency Constraints.** Motivated by the findings in Section 3, EvidenT employs a *Dependency*
516 *Constraints* tool to expose dependency and configuration-related constraints that frequently govern
517 system-level build outcomes. In Linux distributions, these constraints are primarily encoded in
518 the package *build recipe*, which specifies dependencies, macros, flags, and staged build commands;
519 for RPM-based packaging, this recipe is implemented as a `.spec` file for RPM-based packaging to
520 define dependencies, macros, and staged build commands. The tool systematically parses metadata
521 fields and macro definitions while segmenting build directives into discrete recipe stages such as
522 `%prep`, `%build`, `%install`, and `%check`. By returning these extracted constraints in a structured
523 JSON format, the tool enables the LLM to reason explicitly about ISA-dependent build requirements
524 and configuration flaws. This approach provides structured evidence that eliminates the need for
525 the agent to infer complex build logic implicitly from raw text.
526 **Structure Localization.** To complement distilled failure signals and dependency constraints,
527 EvidenT employs a *Structure Localization* tool that provides a global view of the package organi-
528 zation. Given the package workspace, the tool traverses the directory tree, prunes non-essential
529 artifacts such as documentation and generated files, and returns a hierarchical inventory of source
530 files and configuration artifacts. This structural evidence highlights plausible modification targets
531 and cross-file relationships, thereby improving localization precision in complex, multi-language
532 system-level packages.
533
534 *4.3.2 System-level Repair Actions.* After analysis localizes failure-relevant artifacts, EvidenT enters
535 the *Repair* phase and applies system-level repair actions. Unlike traditional program repair that
536 focuses on source code alone, system-level package repair often requires coordinated updates
537 across heterogeneous artifacts, including packaging recipes, configuration scripts, and compressed
538 source archives. Guided by the iteration-aware evidence context maintained by the controller, the
539

orchestrator selects and applies one of the following repair actions, with all edits recorded for subsequent validation and history-aware reasoning.

**Archive Repackaging.** System-level packages are commonly distributed as compressed source archives. To support end-to-end repair, EvidenT performs round-trip archive editing by unpacking the source archive into a temporary workspace, applying targeted edits, and reassembling the archive in the expected format for the build service. To prevent invalid submissions, EvidenT enforces two correctness constraints: the extracted workspace is never uploaded directly, and any modified content must be re-archived before validation. This design ensures that each iteration is evaluated under realistic packaging conditions and remains compatible with production build workflows.

**Configuration Adaptation.** EvidenT supports configuration adaptation on the build recipe and related packaging scripts, including dependency declarations, macro definitions, compiler or linker flags, and stage-specific directives. Edits are applied through a full-file replacement policy to avoid partial or inconsistent changes, and each modification is recorded with its iteration index to support history-aware refinement.

**Source Modification.** When configuration adaptation is insufficient, EvidenT applies source modification to address code-level incompatibilities, missing definitions, or architecture- and toolchain-dependent assumptions. The system first retrieves the complete content of a target file for inspection, then overwrites it with a revised version as a single atomic update. For traceability, EvidenT records a diff-style edit summary for each modification and appends it to the repair history, preventing redundant or conflicting edits in later iterations.

*4.3.3 Build-based Validation Tools.* As shown in Figure 1, the *Build-based Validation Tools* implement the *Validation* phase of EvidenT and close the repair loop by rebuilding the patched package under a reproducible environment and returning the resulting failure evidence to the controller.

**External Build Invocation.** After a candidate fix is applied, EvidenT submits the reconstructed package to the external build service through an upload-and-trigger interface. To preserve a strict iteration boundary, each repair iteration issues exactly one build invocation, ensuring that all candidate fixes are evaluated under the same clean and reproducible conditions.

**Result-based Validation.** During execution, EvidenT monitors the submitted build until it reaches a terminal state. To avoid indefinite waiting, the validator enforces a maximum monitoring window of 600 seconds, calibrated to typical build completion times observed in our package set. If the build does not terminate within this window, EvidenT marks the attempt as a timeout and treats it as an unsuccessful iteration. A timeout differs from a normal failure in that the build service may not expose a complete new build log when the monitoring window expires. In this case, the validator produces *partial* feedback that records the timeout outcome and the latest observable build status returned by the build service. For non-timeout failures, the validator retrieves the newly generated build log and stores it locally. In both cases, the resulting build feedback is forwarded to the Evidence-Preserving Repair Controller to initiate the next iteration.

**Iterative Feedback Loop.** By coupling external rebuilds with outcome-aware validation, EvidenT forms a build-and-feedback loop. Each iteration either terminates with a verified successful build or yields updated failure evidence that triggers the next round of localization and repair.

## 5 Evaluation

We evaluate EvidenT on real-world system-level package build failures to answer the following research questions:

- **RQ3 (Effectiveness).** How effective is EvidenT at repairing system-level package build failures?
- **RQ4 (Generalization).** Does EvidenT generalize across different ISAs?

589 • **RQ5 (Iteration Necessity).** To what extent are iterative repair loops necessary?
590 • **RQ6 (Ablation).** How do orchestration-layer components affect repair performance?
591 • **RQ7 (Failure Analysis).** Why does EvidenT fail on certain packages, and what limitations do
592 these failures reveal?
593
594
### 5.1 Experimental Setup

595 **Dataset.** We collect 219 RISC-V, 79 aarch64, and 83 x86_64 build-failed packages from the Open
596 Build Service (OBS). These packages feature heterogeneous artifacts such as recipes, scripts, and
597 source archives alongside multi-language code. Each package includes build logs capturing multi-
598 stage failure evidence. We reproduce all reported failures before evaluation and use the RISC-V set
599 as our primary benchmark, reserving others for ISA generalization.
600 **LLMs.** We evaluate three LLMs to cover diverse deployment settings. *GPT-5-mini* is selected for its
601 balance of cost-efficiency and reliability in iterative tool-use scenarios. *Qwen3-max* is chosen for its
602 high-capacity orchestration of complex multi-step tasks , while *DeepSeek-v3* serves as a powerful
603 open-source representative with specialized code-oriented performance. All models operate in
604 standard chat-generation mode without enabling separate reasoning variants, with the temperature
605 set to 1.0 for all runs.
606 **Baselines.** We compare EvidenT against three classes of baselines to evaluate its relative effective-
607 ness. (1) *Bare LLM.* This baseline represents a non-agentic approach where the model proposes
608 a repair in a single iteration from raw logs and the workspace. All auxiliary tools are disabled.
609 (2) *Prior tool-augmented agents.* We adapt Agentless [42], CXXCrafter [43], and RepairAgent [3]
610 to the system-level repair setting. Since these agents assume source-centric environments, we
611 uniformly extend their inputs to package-level artifacts and removing their inherent restrictions to
612 single-language codebases. We further replace their validation harnesses with our OBS build service.
613 Crucially, we preserve their original agentic workflows and internal tool-use logic to ensure a fair,
614 reproducible comparison. (3) *EvidenT(w/o evidence).* To isolate the impact of our core principle,
615 this variant employs the same tool suite as EvidenT but clears the evidence after each iteration. It
616 specifically evaluates whether tool access alone is sufficient for success without the coordination
617 provided by an iteration-aware controller.
618 **Metric and Implementation.** A repair is considered successful if the rebuilt package reaches the
619 *succeeded* state on OBS; we report the repair success rate as the fraction of successfully rebuilt
620 packages. We implement EvidenT in Python and use GPT-5-mini as the default LLM. Unless
621 otherwise stated, we allow up to three repair iterations per package. Experiments run on Ubuntu
622 22.04 with an Intel Xeon Gold 5416S CPU (64 cores) and 376 GB RAM.
623
624
### 5.2 RQ3: Effectiveness of EvidenT

625 We evaluate the effectiveness of EvidenT on 219 real-world RISC-V package build failures. Table 4
626 summarizes the repair outcomes, repair time, and build time of EvidenT and all baselines.
627 **Overall Repair Effectiveness.** Using GPT-5-mini, EvidenT repairs 118 out of 219 packages,
628 achieving a success rate of **53.88%**. In contrast, bare LLM baselines remain below 2.00% (0.91–
629 1.83%), indicating that single-pass log-to-patch generation without structured tool support rarely
630 succeeds under build-service verification.
631 Among adapted prior agents, effectiveness remains limited. Agentless repairs only 9 packages
632 (4.11%) and generates 196 *B/U* cases. Its file-centric workflow struggles to (i) localize failure-relevant
633 artifacts across complex packaging layouts, (ii) carry forward build feedback and repair history,
634 and (iii) constrain edits under a fixed prompt schema with global rules. These limitations hinder
635 actionable repairs, even after adapting inputs and OBS verification. CXXCrafter achieves 20.55%
636 success. It focuses on generating and refining end-to-end build solutions such as Dockerfiles and
637

Table 4. Comparative evaluation of repair effectiveness and efficiency. Success, Failed, and B/U (Broken/Unsolvable) counts partition the 219-package dataset. B/U denotes cases with missing descriptions or unresolved dependencies. *Time-R* reports the average repair overhead per package (excluding the build phase), while *Time-B* reports the average external build duration in seconds.

| Methodology | Success | Failed | B/U | Success Rate (%) | Time-R (s) | Time-B (s) |
|---|---|---|---|---|---|---|
| Bare LLM (Qwen3-max) | 2 | 190 | 27 | 0.91 | 48.53 | 149.78 |
| Bare LLM (Deepseek-v3) | 3 | 177 | 39 | 1.37 | 44.56 | 134.00 |
| Bare LLM (GPT-5-mini) | 4 | 191 | 24 | 1.83 | 40.38 | 106.75 |
| Agentless [42] (GPT-5-mini) | 9 | 14 | 196 | 4.11 | 189.37 | 208.30 |
| CXXCrafter [43] (GPT-5-mini) | 45 | 41 | 133 | **20.55** | 303.05 | 526.22 |
| RepairAgent [3] (GPT-5-mini) | 13 | 195 | 11 | 5.94 | 176.62 | 219.78 |
| EvidenT(w/o evidence) (GPT-5-mini) | 42 | 172 | 5 | 19.18 | 499.20 | 488.83 |
| EvidenT (Deepseek-v3) | 39 | 157 | 23 | 17.81 | 565.20 | 516.53 |
| EvidenT (Qwen3-max) | 57 | 99 | 63 | 26.03 | 507.82 | 549.78 |
| EvidenT (GPT-5-mini) | **118** | 59 | 42 | **53.88** | 510.16 | 518.75 |

scripts, rather than localizing and repairing package-level artifacts and archived sources. Without history-aware constraints and evidence slots, repairs are limited to compilation symptoms and fail to address packaging and dependency-related issues. RepairAgent repairs 13 packages (5.94%) but lacks effective evidence organization. Failure evidence is not compacted into actionable signatures, unsuccessful edits are not flagged as "do-not-repeat," and cross-artifact localization remains weak. These issues lead to drift and non-convergence under build-based feedback. EvidenT(w/o evidence) repairs 42 packages (19.18%), showing improvement over bare LLMs. However, by discarding cross-iteration evidence, it repeats or introduces inconsistent actions, limiting convergence and further repair success.

**Time Efficiency.** Table 4 reports both repair-side overhead and OBS build duration. Build time is largely determined by external verification and failure stage, while the main runtime differences arise from the repair process itself.

Bare LLM baselines have the shortest repair time but almost never succeed, confirming that lightweight log-only reasoning is insufficient for system-level repair. Among adapted agents, Agentless and RepairAgent exhibit relatively low repair time (189.37 s and 176.62 s). This is consistent with their original design philosophy of keeping the agent workflow lightweight and file-centric, where the agent quickly selects a small set of candidate locations and generates a patch without maintaining iteration-level evidence slots or negative history. CXXCrafter incurs higher repair time (303.05 s), reflecting additional effort spent on refining build procedures and environments. EvidenT(w/o evidence) and EvidenT incur comparable repair time, suggesting that tool-driven analysis and artifact-level edits dominate the repair cost in this setting, while the additional overhead of iteration-aware evidence persistence is limited relative to the end-to-end build-and-verify loop.

**Impact of Failure Category and Software Languages.** We further analyze repair effectiveness across the four failure categories identified in Section 3. EvidenT attains comparable success rates across categories: 68.18% for test failures, 61.90% for dependency failures, 55.56% for compilation failures, and 50.00% for packaging failures. The reason is that category breakdown aligns with the types of evidence and repair targets exposed in package builds. Dependency failures frequently require reasoning over build recipes and dependency constraints, while test failures are largely driven by test-stage evidence in build logs. For compilation and packaging failures, directory structure and retrieved knowledge help map log evidence to concrete modification targets spanning recipes and source archives. We also examine robustness across programming languages. Among

687 Table 5. Preliminary results of EvidenT's generalizability across different Instruction Set Architectures (ISAs).
688
689
| ISA | Methodology | Success | Failed | B/U | Success Rate (%) | Time-R (s) | Time-B (s) |
|---|---|---|---|---|---|---|---|
| aarch64 | Agentless [42] (GPT-5-mini) | 1 | 67 | 11 | 1.27 | 197.42 | 142.01 |
| | CXXCrafter [43] (GPT-5-mini) | 15 | 9 | 55 | 18.99 | 216.99 | 318.31 |
| | RepairAgent [3] (GPT-5-mini) | 4 | 64 | 11 | 5.06 | 212.07 | 176.44 |
| | EvidenT(w/o evidence) (GPT-5-mini) | 22 | 43 | 14 | 27.85 | 550.20 | 333.76 |
| | EvidenT (GPT-5-mini) | **33** | 40 | 6 | **41.77** | 547.63 | 466.21 |
| x86_64 | Agentless [42] (GPT-5-mini) | 3 | 53 | 27 | 3.61 | 181.09 | 219.25 |
| | CXXCrafter [43] (GPT-5-mini) | 9 | 12 | 62 | 10.84 | 208.24 | 317.40 |
| | RepairAgent [3] (GPT-5-mini) | 1 | 70 | 12 | 1.20 | 210.72 | 139.29 |
| | EvidenT(w/o evidence) (GPT-5-mini) | 19 | 51 | 13 | 22.89 | 663.47 | 347.00 |
| | EvidenT (GPT-5-mini) | **39** | 36 | 8 | **46.99** | 525.41 | 504.37 |

690 691 692 693 694 695 696 697 698 699 700

701 the 219 RISC-V packages, 80 involve multi-language mixtures, of which EvidenT repairs 42. For
702 language-dominant packages, EvidenT repairs 54/89 Python packages and 8/20 C/C++ packages,
703 indicating that its repair decisions are primarily conditioned on package-level artifacts and build
704 evidence rather than a single programming language.
705 **Impact of LLM Choice.** Replacing GPT-5-mini with DeepSeek-v3 or Qwen3-max reduces Evi-
706 denT's success rate to 17.81% and 26.03%, respectively. Despite these differences, EvidenT consis-
707 tently outperforms all baselines under each model, indicating that performance gains mainly come
708 from iteration-aware evidence management and tool-orchestrated repair and validation, rather
709 than relying on model capacity alone.
710
711 **Finding 3:** EvidenT achieves a repair success rate of **53.88%** using GPT-5-mini, outperform-
712 ing the strongest baselines. This improvement holds consistently across failure categories
713 (50.00–68.18%) and language settings, including 52.50% success on multi-language packages.
714
715
716 ### 5.3 RQ4: Generalization across Different ISAs

717 To evaluate the architectural robustness of EvidenT, we investigate its ability to generalize beyond
718 the RISC-V ecosystem. A pivotal design principle of EvidenT is the separation of concerns, which
719 decouples the ISA-agnostic repair workflow from the ISA-specific *Knowledge Context*. This archi-
720 tecture ensures that all core modules operate on generic system-level artifacts, including build logs,
721 packaging recipes, and source archives, without being hard-coded for any specific ISA.
722 **Seamless Adaptation.** We instantiated EvidenT for x86_64 and aarch64 by simply swapping the
723 underlying architecture-related data sources in the *Knowledge Context*, as detailed in Section 4.2.
724 As summarized in Table 5, EvidenT achieves a repair success rate of 41.77% (33/79) on aarch64 and
725 46.99% (39/83) on x86_64. The consistency of these results across three distinct ISAs demonstrates
726 that our evidence-preserving principle serves as a universal solution for system-level failures.
727 **Evidence Preservation and Baseline Comparison.** EvidenT consistently outperforms all
728 adapted agent baselines across both architectures by a significant margin. Specifically, Agent-
729 less achieves only 1.27% on aarch64 and 3.61% on x86_64, whereas CXXCrafter and RepairAgent
730 fail to exceed 19% and 6% respectively. This performance gap indicates that tool access alone is
731 insufficient for navigating heterogeneous package ecosystems. Overall, these results show that
732 without structured evidence management, generic agents struggle to interpret and reuse complex
733 failure signals.
734
735

736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784

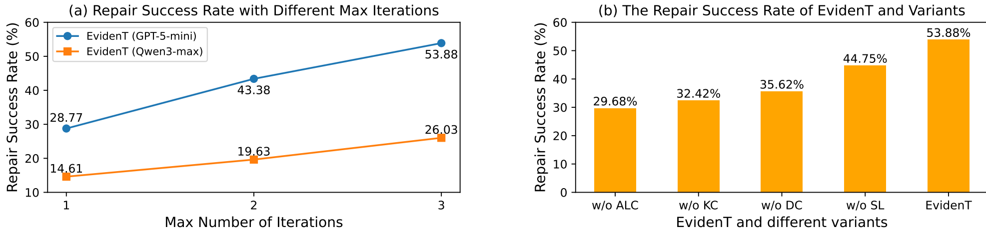



Fig. 3. Repair success rates of EvidenT. (a) Repair success rates under different maximum iteration budgets (1–3) for GPT-5-mini and Qwen3-max. (b) Success rates of ablated variants removing each component.

**Finding 4:** EvidenT repairs 41.77% of aarch64 and 46.99% of x86_64 packages by updating only the ISA-specific Knowledge Context.

## 5.4 RQ5: Iteration Necessity

System-level package repair relies on build-based verification, where each repair attempt is assessed by an external rebuild and the next decision depends on the latest build feedback and prior edits. For each iteration, the Evidence-Preserving Repair Controller refreshes *Build Feedback*, *Repair History*, and *Cached Findings* (Section 4.2) before the subsequent analysis and repair steps. Figure 3(a) reports repair success rates as the maximum number of iterations increases from one to three. With one iteration, EvidenT repairs 63 packages (28.77%) using GPT-5-mini and 32 packages (14.61%) using Qwen3-max. With two iterations, the numbers increase to 95 (43.38%) and 43 (19.63%), respectively. With three iterations, repair success further increases to 118 (53.88%) and 57 (26.03%). These results indicate that additional iterations improve repair effectiveness under build-service feedback.
**Repair Outcome Patterns.** We categorize repair trajectories by the iteration at which they converge. First-iteration successes typically occur when the initial build feedback contains a localized and actionable error signal that can be mapped to a single repair target, such as a build recipe directive or a specific source file. Second-iteration successes often arise when the first attempt addresses part of the failure, and the subsequent rebuild exposes refined evidence, such as an error that shifts to a later build stage or a newly surfaced missing dependency, enabling a more targeted follow-up edit. Non-convergent cases persist when iterative rebuilds fail to provide refinement signals. In particular, some packages terminate early due to corrupted archives, missing build recipes, or invalid build metadata, leaving no reliable target for modification. Other cases produce inconsistent failure signatures across iterations, so updates cannot be narrowed to a stable artifact or stage.

**Finding 5:** Increasing the iteration budget from one to three raises repair success by 25.11 percentage points with GPT-5-mini (28.77% $\longrightarrow$ 53.88%) and by 11.42 points with Qwen3-max (14.61% $\longrightarrow$ 26.03%).

## 5.5 RQ6: Contribution of Analysis & Repair Orchestration Components

To evaluate the necessity of the evidence-preservation principle, we perform an ablation study by removing one key component at a time. We compare the full EvidenT against four variants: (i) $\text{EvidenT}_{\text{w/o ALC}}$, which disables Anomaly-focused Log Condensation; (ii) $\text{EvidenT}_{\text{w/o KC}}$, removing Knowledge Context retrieval; (iii) $\text{EvidenT}_{\text{w/o DC}}$, omitting Dependency Constraints; and (iv) $\text{EvidenT}_{\text{w/o SL}}$, bypassing Structure Localization.

**EvidenT$_{w/o\ ALC}$.** Disabling ALC causes the most significant performance regression, with the success rate plummeting from 53.88% to 29.68%. Without this stage, build feedback is no longer distilled into compact, phase-aligned failure signatures. Therefore, subsequent analysis cannot reliably prioritize salient signals from noisy logs and becomes more prone to leading to misaligned and ineffective edits.
**EvidenT$_{w/o\ KC}$.** Removing *Knowledge Context* retrieval reduces the success rate to 32.42%. This mainly affects cases where the repair depends on ISA or toolchain-specific conventions not explicit in the current build log. The agent can no longer leverage matched historical experience, facing higher ambiguity.
**EvidenT$_{w/o\ DC}$.** The absence of Dependency Constraints lowers the success rate to 35.62%. Without structured recipe-level constraints, the agent must recover dependencies, macros, and stage directives from raw text, which increases incomplete or inconsistent configuration edits.
**EvidenT$_{w/o\ SL}$.** Removing Structure Localization yields a smaller but consistent decrease to 44.75%. The structural inventory helps map localized signals to concrete edit candidates in large workspaces. Without it, target selection becomes less stable for multi-component packages.

**Finding 6:** Removing any component reduces repair success from 53.88% to 29.68–44.75%, indicating that effective repair in EvidenT relies on their complementary contributions.

### 5.6 RQ7: Failure Analysis and Limitations

**Tool Usage Behavior.** We analyze 7,923 tool invocations across 219 packages. Tool calls largely follow the intended analysis–repair–validation workflow, with only three cases exhibiting misordered execution. For 34 packages, the repair process terminates early before completing required validation steps, typically due to corrupted archives, missing build recipes, or invalid build metadata. Overall, unsuccessful repairs are more often associated with insufficient or unusable failure evidence than with systematic tool misuse.
**Limitations.** EvidenT is most effective when build feedback provides stable and actionable refinement signals across iterations. Some packages remain unrepaired when the build service cannot produce complete evidence (e.g., incomplete artifacts or unstable failure traces), preventing the controller from conditioning subsequent iterations on comparable feedback. Handling such cases may require additional recovery support for invalid inputs and partial build feedback.

**Finding 7:** Among 7,923 tool invocations, misordered execution is rare (3 cases), while 34 packages fail due to early termination under invalid or incomplete build artifacts.

### 5.7 Case Study

We present a case study on `postquantumcryptoengine` [33], a C++ library for post-quantum cryptography, to illustrate how EvidenT performs evidence-driven system-level repair under complex artifact and dependency conditions. The initial build on `riscv64` fails during compilation with unresolved template symbols and missing type definitions related to the Open Quantum Safe (OQS) library.

Using *Failure Signal Extraction*, EvidenT condenses the raw build log into a small set of architecture-tagged error signatures, which indicate that the failure originates from C++ header processing rather than linker configuration. In parallel, *Dependency Constraints* analysis over the spec file exposes strict version and feature requirements on `liboqs`, ruling out missing-package installation as the primary cause. To identify the concrete faulty artifact, EvidenT applies *Structure Localization* to inspect the package workspace and source archive layout. This step narrows the failure to a

834 truncated header file, `crypto.hh`, whose class declarations are incomplete and lack required OQS
835 includes, preventing correct instantiation of cryptographic components. Guided by this localized ev-
836 idence, EvidenT reconstructs the missing class hierarchy and restores the expected OQS interfaces
837 through a targeted source-level modification. After repackaging the modified archive, EvidenT
838 validates the fix through the external build service. Two intermediate iterations expose residual
839 compilation errors, which are resolved by refining the header definitions based on updated build
840 feedback. The third iteration completes successfully, yielding a clean rebuild.
841
842 ## 6 Discussion

843 **Design Trade-offs and Remaining Failure Modes.** EvidenT prioritizes evidence-driven and
844 minimally repairs to balance immediate success with long-term maintainability, as aggressive
845 modifications often undermine build reproducibility. For some unresolved cases, failures are often
846 constrained by ecosystem-level factors rather than improper tool orchestration alone. In particular,
847 incomplete or corrupted source artifacts, unstable dependency availability, and non-deterministic
848 build or test behaviors may prevent convergence even when failure localization is accurate. These
849 observations suggest that repair efficacy is bounded by both agent reasoning and the quality of the
850 surrounding build infrastructure.
851 **Build Success vs. Functional Correctness.** A successful build does not necessarily guarantee
852 full functional correctness at runtime. Our evaluation focuses on build-level success as a practical
853 and reproducible proxy for large-scale system-level repair. While this metric captures the abil-
854 ity to resolve concrete build failures, additional validation (e.g., regression testing, performance
855 benchmarking, and integration testing) is required to assess post-build functionality. Incorporating
856 systematic runtime testing into EvidenT remains an important direction for future work.
857 **Threats to Validity.** Several factors may affect the validity of our results. First, repair performance
858 can vary across LLMs due to differences in tool-use reliability and code generation capability, as
859 well as the inherent stochasticity of model outputs. To mitigate this effect, we repeat all experiments
860 twice using the default model (GPT-5-mini) and observe no noticeable variation in repair success.
861 Second, our evaluation relies on the Open Build Service (OBS) for build-based validation, whose
862 updates or instability may affect consistency. We plan to address this limitation by adopting
863 containerized build environments (e.g., Docker) to further improve reproducibility.
864
865 ## 7 Conclusion

866
867 In this paper, we present EvidenT, an evidence-preserving framework for iterative system-level
868 package repair. We conduct an empirical study of real-world package build failures to characterize
869 their diverse root causes across heterogeneous artifacts, build stages, and validation environments.
870 Based on these observations, EvidenT integrates coordinated tool support for failure analysis,
871 artifact-level repair, and build-based validation, enabling repairs to be guided by accumulated
872 evidence over multiple iterations. Our evaluation on real package failures across multiple instruc-
873 tion set architectures demonstrates the effectiveness and portability of EvidenT. Specifically,
874 EvidenT repairs 118/219 packages on RISC-V (53.88%), 33/79 on aarch64 (41.77%), and 39/83 on
875 x86_64 (46.99%), substantially outperforming bare LLM baselines and representative agent-based ap-
876 proaches. These results highlight the importance of preserving and reusing failure evidence, as well
877 as validating repairs through external build services, for scalable automated package maintenance.
878
879 ## 8 Data Availability

880 We have made our source code and datasets publicly available through an anonymous repository
881 at https://anonymous.4open.science/r/EvidenT-1638/README.md.
882

883 ## References


884 [1] Henri Aïdasso, Mohammed Sayagh, and Francis Bordeleau. 2025. Build Optimization: A Systematic Literature Review. *Comput. Surveys* 58, 2 (2025), 1–38. doi:10.1145/3757912
885
886 [2] Saikat Barua. 2024. Exploring autonomous agents through the lens of large language models: A review. *arXiv preprint arXiv:2404.04442* (2024). https://arxiv.org/abs/2404.04442
887
888 [3] Islem Bouzenia, Prem Devanbu, and Michael Pradel. 2025. RepairAgent: An Autonomous, LLM-Based Agent for Program Repair. In *Proceedings of the International Conference on Software Engineering (ICSE).* Research Track; ArXiv preprint arXiv:2403.17134.
889
890 [4] Bihuan Chen, Hongyu Zhang, Zhenchang Zhou, Chang Xu, and Baowen Xu. 2021. BuildFast: History-Aware Build Outcome Prediction for Fast Build Triage. In *Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering (ASE).* IEEE, 1025–1037.
891
892 [5] Yinfang Chen, Minghua Ma, Huaibing Xie, Yu Kang, Xin Gao, Xuchao Zhang, Liu Shi, Yunjie Cao, Xuedong Gao, Hao Fan, Ming Wen, Jun Zeng, Saravan Rajmohan, Dongmei Zhang, and Tianyin Xu. 2024. Large Language Models Can Provide Accurate and Interpretable Incident Triage. In *2024 IEEE 35th International Symposium on Software Reliability Engineering (ISSRE).* IEEE.
893
894
895
896 [6] Yinfang Chen, Huaibing Xie, Minghua Ma, Yu Kang, Xin Gao, Liu Shi, Yunjie Cao, Xuedong Gao, Hao Fan, Ming Wen, Jun Zeng, Supriyo Ghosh, Xuchao Zhang, Chaoyun Zhang, Qingwei Lin, Saravan Rajmohan, Dongmei Zhang, and Tianyin Xu. 2024. Automatic Root Cause Analysis via Large Language Models for Cloud Incidents. In *Proceedings of the 19th European Conference on Computer Systems (EuroSys).* ACM, 674–688.
897
898
899 [7] Jürgen Cito and H. C. Gall. 2016. Using Docker Containers to Improve Reproducibility in Software Engineering Research. *Proceedings of the 38th International Conference on Software Engineering* (2016), 1–10. doi:10.1145/2889160.2891057
900
901 [8] Enfang Cui, Tianzheng Li, and Qian Wei. 2023. Risc-v instruction set architecture extensions: A survey. *IEEE Access* 11 (2023), 24696–24711.
902
903 [9] Gang Fan, Chengpeng Wang, Rongxin Wu, Xiao Xiao, Qingkai Shi, and Charles Zhang. 2020. Escaping dependency hell: finding build dependency errors with the unified dependency graph. In *ISSTA '20: 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, USA, July 18-22, 2020*, Sarfraz Khurshid and Corina S. Pasareanu (Eds.). ACM, 463–474. doi:10.1145/3395363.3397388
904
905
906 [10] Fedora Project. 2025. Koji is an RPM-based build system used by the Fedora Project and others. https://koji.build/. Accessed: 2025-09-01.
907
908 [11] Blake W Ford, Apan Qasem, Jelena Tešić, and Ziliang Zong. 2021. Migrating software from x86 to ARM Architecture: An instruction prediction approach. In *2021 IEEE International Conference on Networking, Architecture and Storage (NAS).* IEEE, 1–6.
909
910 [12] Apache Software Foundation. 2025. Apache Maven. https://maven.apache.org
911 [13] Python Software Foundation. 2025. pip: The Python Package Installer. https://pip.pypa.io
912 [14] Ryan Gibb, Patrick Ferris, David Allsopp, Michael Winston Dales, Mark Elvers, Thomas Gazagnaire, Sadiq Jaffer, Thomas Leonard, Jon Ludlam, and Anil Madhavapeddy. 2025. Solving Package Management via Hypergraph Dependency Resolution. *arXiv preprint arXiv:2506.10803* (2025). https://arxiv.org/abs/2506.10803
913
914 [15] Foyzul Hassan, Shaikh Mostafa, Edmund S. L. Lam, and Xiaoyin Wang. 2017. Automatic Building of Java Projects in Software Repositories: A Study on Feasibility and Challenges. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE).* IEEE, 379–389. doi:10.1109/ASE.2017.8115651
915
916
917 [16] Foyzul Hassan and Xiaoyin Wang. 2018. HireBuild: an automatic approach to history-driven repair of build scripts. In *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, Michel Chaudron, Ivica Crnkovic, Marsha Chechik, and Mark Harman (Eds.). ACM, 1078–1089. doi:10.1145/3180155.3180181
918
919
920 [17] Md Hassan, Tao Wang, Shaowei Wang, and David Lo. 2019. Predicting Build Failures Using Social Network Analysis on Developer Communication. In *Proceedings of the 41st International Conference on Software Engineering (ICSE).* ACM, 120–130.
921
922 [18] Minghua He, Tong Jia, Chiming Duan, Huaqian Cai, Ying Li, and Gang Huang. 2024. LLMeLog: An Approach for Anomaly Detection based on LLM-enriched Log Events. In *2024 IEEE 35th International Symposium on Software Reliability Engineering (ISSRE).* 132–143. doi:10.1109/ISSRE62328.2024.00023
923
924
925 [19] Shilin He, Pinjia He, Zhuangbin Chen, Tianyi Yang, Yuxin Su, and Michael R. Lyu. 2021. A Survey on Automated Log Analysis for Reliability Engineering. 54, 6, Article 130 (July 2021), 37 pages. doi:10.1145/3460345
926
927 [20] Jordan Henkel, Denini Silva, Leopoldo Teixeira, Marcelo d'Amorim, and Thomas W. Reps. 2021. Shipwright: A Human-in-the-Loop System for Dockerfile Repair. In *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021.* IEEE, 1148–1160. doi:10.1109/ICSE43902.2021.00106
928
929 [21] Xinyi Hou, Yanjie Zhao, Shenao Wang, and Haoyu Wang. 2025. Model Context Protocol (MCP): Landscape, Security Threats, and Future Research Directions. arXiv:2503.23278 [cs.CR] https://arxiv.org/abs/2503.23278
930
931

932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980


[22] Lars Huning and Elke Pulvermueller. 2021. Automatic Code Generation of Safety Mechanisms in Model-Driven Development. *Electronics* 10, 24 (2021), 3150. https://www.mdpi.com/2079-9292/10/24/3150

[23] 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.

[24] Kitware. 2025. CMake: Cross-Platform Make. https://cmake.org

[25] Naveen Krishnan. 2025. Advancing Multi-Agent Systems Through Model Context Protocol: Architecture, Implementation, and Applications. https://arxiv.org/html/2504.21030v1. Accessed: 2025-09-01.

[26] 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). https://arxiv.org/abs/2308.03688

[27] Minghua Ma, Yinfang Chen, Huaibing Xie, Xuchao Zhang, Yu Kang, Xin Gao, Liu Shi, Yunjie Cao, Hao Fan, Ming Wen, Saravan Rajmohan, Dongmei Zhang, and Tianyin Xu. 2024. MonitorAssistant: Simplifying Cloud Service Monitoring via Large Language Models. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE)*. ACM.

[28] C. Macho. 2024. DValidator: An approach for validating dependencies in build scripts. *Journal of Systems and Software* 195 (2024), 111916. doi:10.1016/j.jss.2023.111916

[29] Ching Hang Mak and Shing-Chi Cheung. 2024. Automatic build repair for test cases using incompatible Java versions. *Inf. Softw. Technol.* 172 (2024), 107473. doi:10.1016/J.INFSOF.2024.107473

[30] Jordan Matelsky, Gregory Kiar, Erik Johnson, Corban Rivera, Michael Toma, and William Gray-Roncal. 2018. Container-Based Clinical Solutions for Portable and Reproducible Image Analysis. *Journal of Digital Imaging* 31, 3 (2018), 315–320. doi:10.1007/s10278-018-0089-4

[31] D. Moreau and K. Wiebels. 2021. Containers for Computational Reproducibility. *Nature Computational Science* 1, 1 (2021), 1–10. doi:10.1038/s41599-020-00661-w

[32] Olivier Nourry, Yutaro Kashiwa, Weiyi Shang, Honglin Shu, and Yasutaka Kamei. 2025. My Fuzzers Won't Build: An Empirical Study of Fuzzing Build Failures. *ACM Trans. Softw. Eng. Methodol.* 34, 2 (2025), 29:1–29:30. doi:10.1145/3688842

[33] openSUSE Project. 2025. postquantumcryptoengine — openSUSE:Factory. https://build.opensuse.org/package/show/openSUSE:Factory/postquantumcryptoengine.

[34] CMU SEI. 2025. Vessel: Reproducible Container Builds. https://www.sei.cmu.edu/documents/6315/Vessel_Fact_Sheet_TtatchC.pdf

[35] Hyunsook Seo, Ahmed E. Hassan, and Michael W. Godfrey. 2021. Code Review of Build System Specifications: Prevalence, Purposes, Patterns, and Perceptions. In *Proceedings of the 43rd International Conference on Software Engineering (ICSE)*. ACM, 549–560.

[36] Hyunsook Seo, Ahmed E. Hassan, and Michael W. Godfrey. 2022. Understanding the Implications of Changes to Build Systems. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. ACM, 1043–1054.

[37] Usman Shahid. 2025. LLM Tool Calling Series [Part 1]: Understanding Tool Calling and the Model Context Protocol (MCP). https://usmanshahid.medium.com/llm-tool-calling-series-part-1-understanding-tool-calling-and-the-model-context-protocol-mcp-911a7c422fd8. Accessed: 2025-09-01.

[38] Manish Shetty, Yinfang Chen, Gagan Somashekar, Minghua Ma, Yogesh Simmhan, Xuchao Zhang, Jonathan Mace, Dax Vandevoorde, Pedro Las-Casas, Shachee Mishra Gupta, Suman Nath, Chetan Bansal, and Saravan Rajmohan. 2024. Building AI Agents for Autonomous Clouds: Challenges and Design Principles. In *Proceedings of the 15th ACM Symposium on Cloud Computing (SoCC)*. ACM.

[39] Gengyi Sun. 2025. Intelligent Automation for Accelerating the Repair of Software Build Failures. In *47th IEEE/ACM International Conference on Software Engineering, ICSE 2025 - Companion Proceedings, Ottawa, ON, Canada, April 27 - May 3, 2025*. IEEE, 205–207. doi:10.1109/ICSE-COMPANION66252.2025.00062

[40] 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 preprint arXiv:2305.14825. doi:10.48550/arXiv.2305.14825 Version v1 posted 24 May 2023; updated to v2 on 8 Jun 2023.

[41] Huiyan Wang, Lingyu Zhang, Yifan Wu, Xi Xu, Yinxing Liu, Tian Zhang, Lin Zhang, and Hong Mei. 2023. Automatically Resolving Dependency-Conflict Building Failures via Behavior-Consistent Loosening of Library Version Constraints. In *Proceedings of the 31st ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '23)*. Association for Computing Machinery, San Francisco, CA, USA, 707–719. doi:10.1145/3611643.3616309

[42] Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. 2025. Demystifying LLM-Based Software Engineering Agents. *Proc. ACM Softw. Eng.* 2, FSE, Article FSE037 (June 2025), 24 pages. doi:10.1145/3715754

[43] 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. *Proceedings of the ACM on Software Engineering* 2, FSE

(June 2025), 2618–2640. doi:10.1145/3729386

[44] 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 preprint arXiv:2405.10121 [cs.CL]* (2024). https://arxiv.org/abs/2405.10121

[45] Chen Zhang, Bihuan Chen, Junhao Hu, Xin Peng, and Wenyun Zhao. 2022. BuildSonic: Detecting and Repairing Performance-Related Configuration Smells for Continuous Integration Builds. In *37th IEEE/ACM International Conference on Automated Software Engineering, ASE 2022, Rochester, MI, USA, October 10-14, 2022.* ACM, 18:1–18:13. doi:10.1145/3551349.3556923

[46] Lecheng Zheng, Zhengzhang Chen, Jingrui He, and Haifeng Chen. 2024. MULAN: Multi-modal Causal Structure Learning and Root Cause Analysis for Microservice Systems. In *Proceedings of the ACM Web Conference 2024* (Singapore, Singapore) *(WWW '24)*. Association for Computing Machinery, New York, NY, USA, 4107–4116. doi:10.1145/3589334.3645442