Can Language Models Go Beyond Coding?
Assessing the Capability of Language Models to Build Real-World Systems

CHENYU ZHAO, Nankai University, China
SHENGLIN ZHANG[*†‡], Nankai University, China
ZESHUN HUANG, Nankai University, China
WEILIN JIN, Peking University, China
YONGQIAN SUN[§], Nankai University, China
DAN PEI, Tsinghua University, China
CHAOYUN ZHANG, Microsoft, China
QINGWEI LIN, Microsoft, China
CHETAN BANSAL, Microsoft, USA
SARAVAN RAJMOHAN, Microsoft, USA
MINGHUA MA, Microsoft, USA

Large language models (LLMs) have shown growing potential in software engineering, yet few benchmarks evaluate their ability to repair software during migration across instruction set architectures (ISAs). Cross-ISA migration, such as between `x86_64` and `aarch64`, requires handling complex dependencies, heterogeneous toolchains, and long build logs while ensuring executable verification. To address this challenge, we present *Build-bench* [1], an end-to-end benchmark that systematically evaluates the capability of LLMs to repair build failures in cross-ISA settings. *Build-bench* collects 268 real-world failed packages and integrates auxiliary tools including *Structure Extraction*, *File Content Extraction*, *Content Modification*, and *Build Verification* to support autonomous, tool-augmented reasoning. The repair process operates in an iterative loop where, upon failure, the model receives updated build logs and previous repair outcomes to refine subsequent attempts. Through a comparative evaluation across the studied models, *Build-bench* reveals that current models achieve a maximum build success rate of 63.19% and tool usage patterns differ significantly across models. By coupling real build environments with verifiable outcomes, *Build-bench* establishes the first architecture-aware benchmark for studying LLM-based software build and repair.

[*]Corresponding author.
[†]Also with Haihe Laboratory of Information Technology Application Innovation.
[‡]Also with Key Laboratory of Data and Intelligent System Security, Ministry of Education.
[§]Also with Tianjin Key Laboratory of Software Experience and Human Computer Interaction.
[1]Homepage of *Build-bench*: https://aiops-lab-nku.github.io/Build-bench/, Code at https://github.com/zcyyc/Build-bench

Authors' Contact Information: Chenyu Zhao, zhaochenyu@mail.nankai.edu.cn, Nankai University, Tianjin, China; Shenglin Zhang, zhangsl@nankai.edu.cn, Nankai University, Tianjin, China; Zeshun Huang, 2213900@mail.nankai.edu.cn, Nankai University, Tianjin, China; Weilin Jin, 2401112012@stu.pku.edu.cn, School of Computer Science, Peking University, Beijing, China; Yongqian Sun, sunyongqian@nankai.edu.cn, Nankai University, Tianjin, China; Dan Pei, peidan@tsinghua.edu.cn, Tsinghua University, Beijing, China; Chaoyun Zhang, chaoyun.zhang@microsoft.com, Microsoft, Beijing, China; Qingwei Lin, qlin@microsoft.com, Microsoft, Beijing, China; Chetan Bansal, chetanb@microsoft.com, Microsoft, Redmond, USA; Saravan Rajmohan, saravan.rajmohan@microsoft.com, Microsoft, Redmond, USA; Minghua Ma, minghuama@microsoft.com, Microsoft, Seattle, USA.

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.
© 2018 Copyright held by the owner/author(s). Publication rights licensed to ACM.
Manuscript submitted to ACM

CCS Concepts: • **Software and its engineering** → **Software maintenance tools**; **Software configuration management and version control systems**; **Automatic programming**; *Empirical software validation*; • **Computing methodologies** → *Intelligent agents*; • **General and reference** → *Evaluation*.

Additional Key Words and Phrases: Large Language Models, Benchmark, Instruction Set Architecture (ISA), Cross-ISA Migration, Automated Build Repair, Model Context Protocol

ACM Reference Format:
Chenyu Zhao, Shenglin Zhang, Zeshun Huang, Weilin Jin, Yongqian Sun, Dan Pei, Chaoyun Zhang, Qingwei Lin, Chetan Bansal, Saravan Rajmohan, and Minghua Ma. 2018. Can Language Models Go Beyond Coding? Assessing the Capability of Language Models to Build Real-World Systems. In *Proceedings of Make sure to enter the correct conference title from your rights confirmation email (Conference acronym 'XX)*. ACM, New York, NY, USA, 39 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Large language models (LLMs) have become increasingly integrated into modern software development ecosystems, driving remarkable progress in the automation of diverse software engineering tasks. Within the field of software engineering, these LLMs now assist with code generation [31], test synthesis [38], project debugging [29], error detection and issue resolution [35], which together substantially improve developer productivity [30].

To systematically evaluate the strengths and limitations of LLMs in programming contexts, researchers have developed a series of benchmarks that target different aspects of software intelligence. CoderEval [71] focuses on functional code generation, OpenRCA [68] benchmarks LLMs for intelligent root-cause analysis in real operational environments, and SWE-bench [32] assesses end-to-end issue resolution on real-world repositories. Recent extensions such as SWE-bench-Live [76] further enable dynamic evaluation on continuously updated repositories, providing deeper insight into the robustness of autonomous code repair systems. However, these benchmarks primarily focus on functionally-defined repair tasks under **homogeneous software and hardware environments.** They assume that code execution semantics remain consistent across platforms, and that repair success can be verified through test cases or oracle assertions. In contrast, real-world software ecosystems increasingly span **heterogeneous computing architectures**, introducing fundamental differences that challenge this assumption.

Driven by the demand for energy-efficient, cloud-native, and heterogeneous systems, the global computing landscape is undergoing a large-scale transition in instruction set architectures (ISAs) [66]. Among the major ISAs, **x86_64** and **aarch64 (ARM64)** dominate modern computing yet differ substantially in register organization, memory models, compiler behaviors, and toolchains. This divergence has led to widespread cross-ISA migration efforts. Apple's transition to ARM-based M-series chips [2], Amazon's deployment of Graviton processors [1], and Microsoft's ARM-compatible Windows platforms exemplify this shift. Ensuring that large-scale open-source software ecosystems remain portable and buildable across such heterogeneous environments has thus become an urgent challenge for sustainable software evolution.

To maintain correctness and portability during migration between x86_64 and aarch64 platforms, large software ecosystems (*e.g.,* operating systems, middleware, and package repositories) require extensive source-level refactoring and repair. Despite several industrial efforts toward cross-ISA migration [1–3], they rely primarily on internal and ad-hoc evaluation processes, which hinder consistent performance comparison and reproducibility across systems. A major obstacle to automating this process lies in the inherent heterogeneity of cross-ISA build failures, which span multiple stages from environment configuration to packaging (as analyzed in Table 1). Although prior work has explored build repair, diagnosis, and feedback-driven recovery, existing approaches typically provide only localized

or stage-specific support, such as history-driven script repair, dependency-oriented diagnosis, or static root-cause prediction [19, 25, 46, 74, 75]. Such methods struggle with the non-linear nature of migration failures, where the effective repair sequence (*e.g.,* coordinating structure extraction, dependency adjustment, and macro modification) cannot be fully pre-defined. Addressing these failures requires an orchestrator capable of dynamically navigating the tool space and synthesizing heterogeneous feedback from an executable build environment. However, the software engineering community still lacks a benchmark for systematically evaluating whether large language models (LLMs) can understand, adapt, and repair software packages across heterogeneous ISAs.

To bridge this gap, we propose *Build-bench*, the first benchmark designed to evaluate whether LLMs can interpret build-failure contexts, generate effective repairs, and achieve successful rebuild during cross-ISA software packages migration. Constructing such a benchmark introduces multiple challenges.

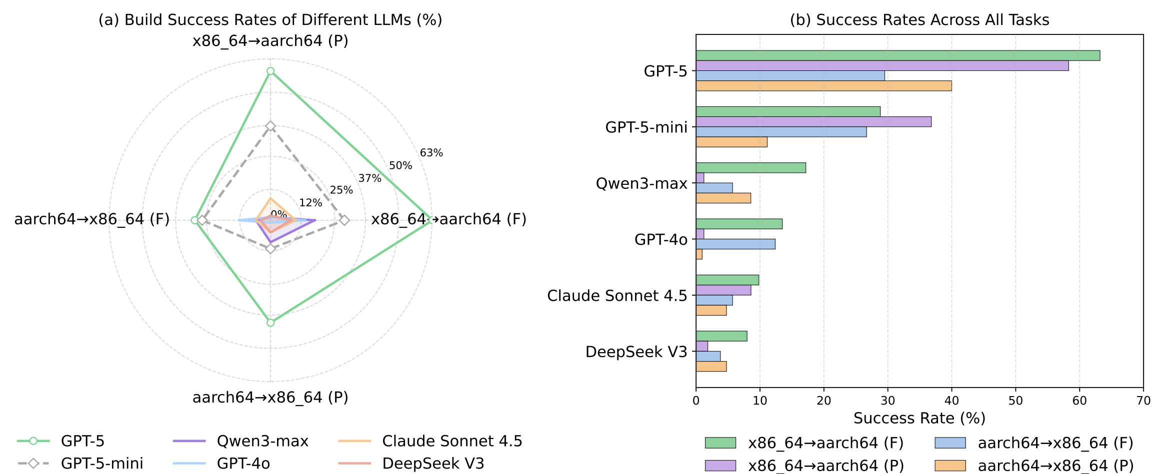

Fig. 1. Comparison of different large language models (LLMs) in cross-ISA build repair tasks. (a) shows the success rates (%) achieved on four migration scenarios (`x86_64→aarch64 (F)`, `x86_64→aarch64 (P)`, `aarch64→x86_64 (F)`, and `aarch64→x86_64 (P)`), where F denotes *Full File Generation* and P denotes *Patch Generation*. (b) summarizes the overall success rates across all tasks for each model.

- **Challenge 1: Capturing the complex reasoning context underlying multi-layer builds.** Cross-ISA build failures rarely occur within a single source file; instead, they arise from intricate dependencies among specification descriptors that define metadata, dependencies, and architecture-specific conditions, together with build scripts, compiler options, and heterogeneous toolchains. Existing benchmarks [32, 76, 81] focus on single-architecture settings and typically provide only file-level inputs with test-based oracles, which are insufficient for evaluating reasoning over such system-level interactions.
- **Challenge 2: Addressing large-scale codebases and complex environments in cross-ISA migration.** On average, each package contains 366 files and over 55,000 lines of code, spanning multiple programming languages (*e.g.,* C/C++, Rust, Python, JavaScript), shell scripts, Makefiles, and architecture-specific configuration files. During cross-ISA migration, the model must not only modify the relevant components but also maintain global consistency across source files, dependency declarations, compiler toolchains, and environment variables.

Such large-scale, multi-language, and architecture-sensitive characteristics fundamentally distinguish this task from small-scale, single-file program repair benchmarks.

- **Challenge 3: Achieving verifiable, end-to-end evaluation.** Most existing benchmarks evaluate LLMs in a single-turn setting, where one repair attempt determines success. In contrast, cross-ISA software package migration relies heavily on iterative feedback. Build logs provide valuable contextual signals that reveal deficiencies in prior modifications and guide progressive refinement. A robust benchmark must assess whether an LLM can leverage such iterative feedback to continuously improve repair accuracy. Moreover, migration success must be validated through executable, end-to-end rebuild rather than textual comparison or test passing. A repair is considered successful only when the package can be fully rebuilt under the target architecture within a controlled, reproducible environment. Consequently, constructing a comprehensive benchmark that integrates iterative reasoning capabilities with verifiable system execution remains a non-trivial challenge.

To address these challenges, *Build-bench* implements an end-to-end evaluation pipeline that integrates automated build verification with iterative reasoning. Following prior work that employs standardized orchestration frameworks such as the Model Context Protocol (MCP) [28], *Build-bench* takes failed build packages (*i.e.,* real packages that succeed on one ISA but fail on another) as input and orchestrates multiple external tools, including *Structure Extraction*, *File Content Extraction*, and source archive *Compression/Decompression*, for package analysis and source manipulation. In the validation phase, *Build-bench* leverages the **Open Build Service (OBS)** [2], which provides a reproducible environment to enable online package building. After the LLM analyzes the failure and proposes minimal modifications based on the system prompt, the repaired software package is automatically uploaded to OBS for rebuilding. *Build-bench* then invokes the *Check Build Result* tool to retrieve build results from OBS. If the build fails, the updated logs and diffs are returned to the LLM for further repair attempts; if it succeeds or the maximum iteration limit is reached, the repair task terminates. This multi-round repair workflow mirrors real industrial migration practices. Through this design, *Build-bench* enables the first systematic evaluation of LLMs on repairing build failures during cross-ISA software migration.

Build-bench consists of 268 real-world software packages, where 163 fail on aarch64 but succeed on x86_64, and 105 in the reverse direction, forming a realistic and challenging corpus of cross-ISA migration failures. We evaluate six state-of-the-art models, namely *GPT-5* [48], *GPT-5-mini* [48], *GPT-4o* [47], *Claude Sonnet 4.5* [4], *DeepSeek V3* [61], and *Qwen3-max* [60], under both *Full File Generation* and *Patch Generation* repair modes. In the *Full File Generation* mode, once an LLM identifies a file containing errors, it directly generates a complete revised version that replaces the original file. In the *Patch Generation* mode, the LLM specifies fine-grained edits such as additions, deletions, or modifications, and the auxiliary tools automatically apply these changes to the source tree.

Fig. 1 summarizes the success rates of all models across two migration directions and two repair strategies. As shown in Fig. 1 (a), GPT-5 maintains consistently high success rates across all scenarios, reaching 63.19% on the x86_64 $\longrightarrow$ aarch64 by *Full File Generation* in particular. Fig. 1 (b) further aggregates the overall success rates across all tasks, revealing a clear performance hierarchy among models. Although GPT-5 and GPT-5-mini nearly outperform the others, large language models still struggle with large-scale, heterogeneous, and architecture-specific repair tasks. By integrating realistic build feedback and executable verification, *Build-bench* provides a unified foundation for evaluating and enhancing software package repair across multiple ISAs.

In summary, the main contributions of *Build-bench* are as follows:

[2]https://openbuildservice.org/

- **A new benchmark and corpus for cross-ISA build.** We present ***Build-bench***, the first benchmark designed to evaluate large language models (LLMs) in repairing software packages that fail during migration across heterogeneous instruction set architectures (ISAs). The benchmark includes 268 real-world software packages, providing a realistic and reproducible corpus for studying architecture-aware reasoning.
- **An end-to-end evaluation framework with iterative build verification.** *Build-bench* integrates real build environments and automated verification pipelines, enabling executable evaluation through multiple repair iterations. The framework captures both build logs and prior modifications as feedback, allowing systematic assessment of LLMs' ability to refine repair strategies under dynamic build contexts.
- **Comprehensive empirical analysis and quantitative insights.** We evaluate six state-of-the-art LLMs covering both proprietary and open-source (GPT-5, GPT-5-mini, GPT-4o, Claude Sonnet 4.5, DeepSeek V3, and Qwen3-max) across *Full File Generation* and *Patch Generation* modes. Our results, summarized in Fig. 1, reveal significant performance variance among models, expose persistent weaknesses in multi-file reasoning and architecture-specific adaptation, and establish baselines for future research on cross-ISA software repair.

2 Background

2.1 Package Building

Software package building [54] is a critical process in software engineering that automates the transformation of source code, specification files (including metadata, dependencies, and architecture-specific conditions), and build scripts into distributable binary packages. It is widely used in operating system distributions (*e.g.,* openSUSE, Debian, Fedora) [16, 20, 49] and large-scale software ecosystems' continuous integration (CI) pipelines. As software systems continue to grow in scale, package building faces increasing challenges [23, 43, 45] related to dependency complexity, environmental inconsistencies, and reproducibility. To address these issues, researchers have explored reproducible builds and automated dependency management mechanisms to improve build stability and security [52].

In practice, the Open Build Service (OBS), an open-source distributed build and release platform, has been widely adopted. OBS supports multi-architecture builds, automated dependency resolution, isolated build environments, and version tracking, enabling the generation of installable packages for multiple distributions from a single platform and providing scalable support for reproducible builds and continuous delivery. Nevertheless, build failures remain common [41], often caused by missing dependencies, version conflicts, or errors in build scripts. As the scale of open-source projects increases, build logs [9, 26] become larger and more complex, making manual analysis inefficient and error-prone.

To address these challenges, both academia and industry have proposed various approaches for diagnosing and repairing build failures, including log pattern mining [9, 11] and history-based automated repair [51]. Recently, artificial intelligence and large language models [7, 33, 39, 77, 78] (LLMs) have shown promising potential for understanding build logs and performing automated repairs. LLM-based methods [13, 64, 69, 72, 73, 79, 82] can identify error patterns, locate root causes, and generate repair suggestions from complex build logs, offering new opportunities for intelligent and automated software package building.

In summary, while OBS provides a reliable infrastructure for large-scale package building, build failures and their repair remain major obstacles to efficiency. Leveraging intelligent techniques [27, 57, 57] such as LLMs for automatic analysis and repair of build logs has thus emerged as a key research direction in software engineering automation.

2.2 Cross-instruction Set Architecture Migration

2.2.1 The Concept of ISA. The Instruction Set Architecture (ISA) [6, 44] serves as the abstract model of a computer, defining the set of commands that the processor can execute, including data types, registers, and memory addressing modes. It forms the crucial interface between hardware and software. Among the diverse ISAs, x86_64 and aarch64 (ARM64) [24, 56] have been the subjects of extensive research and deployment, dominating the server and mobile computing landscapes, respectively. The distinctions among ISAs, such as their instruction sets, calling conventions, and memory models, are fundamental sources of challenges in cross-ISA software migration [5].

2.2.2 Cross-ISA Migration. With the rapid development of heterogeneous computing environments [15], software package building is no longer limited to a single hardware architecture. Cross-ISA migration aims to ensure that software packages can be correctly built and executed on different architectures [21, 22] (*e.g.,* x86_64, aarch64), thereby supporting multi-platform deployment and performance optimization. However, due to differences in instruction sets [21], compilation toolchains [36], and dependency ecosystems [17], migration often faces significant challenges, including compilation failures, dependency conflicts, and runtime errors [55, 62].

In open-source ecosystems, comprehensive multi-architecture support is still limited [58]. Developers often need to manually modify build scripts or source code to adapt to the target architecture. Such manual intervention not only increases maintenance costs but also introduces potential errors, leading to frequent build failures during migration.

Existing research has highlighted the risks associated with hardware-specific dependencies during architecture migrations. For instance, Ford et al. and Davi et al. [14, 21] point out that as servers and mobile devices transition to the ARM architecture, packages reliant on x86-specific features like SSE instructions or particular byte ordering are prone to build failures. Similarly, Wressnegger et al. [65] demonstrate in their study "Twice the Bits, Twice the Trouble" that migrating from 32-bit to 64-bit environments introduces pointer size changes, which can affect memory alignment or cause integer overflows, thereby compromising build correctness and runtime behavior. Due to inconsistent dependencies [12], environment configurations [17], or architecture-specific compilation flags, many packages still fail to build successfully. Therefore, enhancing the automation and intelligence of cross-ISA package migration is crucial for reducing maintenance costs, improving build success rates, and supporting the sustainable development of large-scale software ecosystems, making it a key research direction in software engineering and system maintenance [18, 23, 63].

3 *Build-bench*

In this section, we introduce *Build-bench*, a benchmark designed to evaluate large language models (LLMs) on cross-instruction set architecture (cross-ISA) repair and build tasks. Unlike prior benchmarks that focus on source-level bug fixing or single-architecture builds, *Build-bench* centers on real-world software packages that fail during migration between x86_64 and aarch64. It aims to assess whether LLMs can autonomously diagnose build failures, apply targeted code or configuration modifications, and verify the repaired packages through executable rebuild on a real build service.

3.1 Overview

The overall workflow of *Build-bench*, illustrated in Fig. 2, consists of three major stages: (1) *Input & Diagnosis Context*, (2) *LLM-driven Repair Process*, and (3) *Verification & Evaluation*. In the first stage, *Build-bench* collects essential contextual artifacts from each failed package directory to support diagnosis and repair. Specifically, the inputs include:

- **Source archives** (*e.g.,* `.tar.gz`, `.bz2`), which contain the original code base and associated assets.

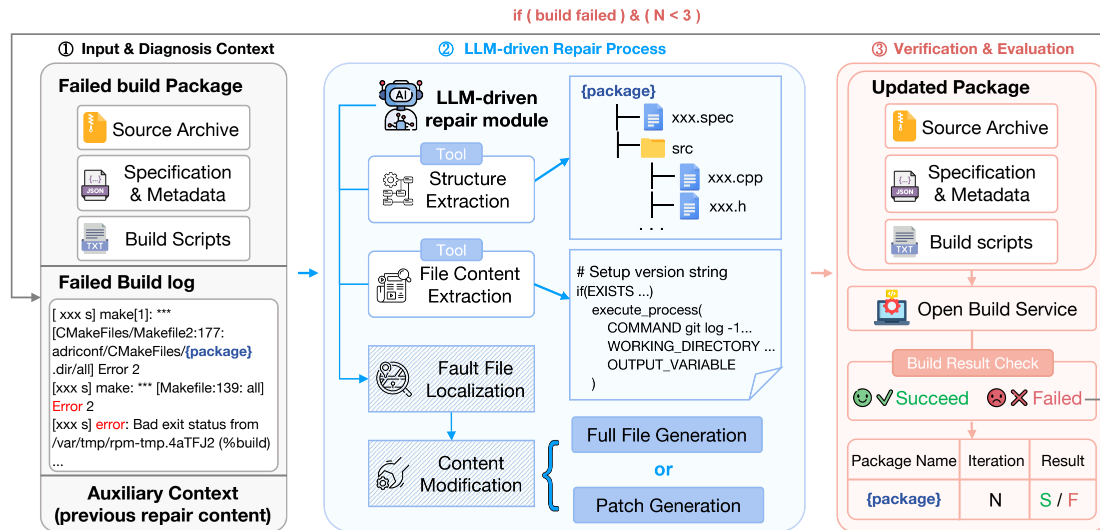

Fig. 2. The automatic cross-ISA repair and build pipeline of *Build-bench*. If the build fails and the maximum iteration $N_{max} = 3$ is not reached, the process repeats with the updated build log as well as the previous repair content.

- **Specification and metadata** (*e.g.*, `.spec`, `.changes`), which define build configurations, dependency declarations, and version updates.
- **Build scripts** (*e.g.*, `.service`, `.desktop`), which provide configuration or service-level integration scripts.
- **Failed build logs**, which records the compiler output and error traces during the failed build.

These inputs and diagnosis contexts provide the diagnostic foundation for failure analysis. For iterations beyond the first, *Build-bench* enriches the inputs with the latest package state, the updated build log, and an **auxiliary context** that preserves modifications from previous attempts, thereby allowing the model to reason with historical information.

The second stage leverages an LLM-driven repair module based on the Model Context Protocol (MCP), which allows dynamic interaction between the model and a suite of external tools for information extraction and content modification. To assess how different editing granularities affect model performance, the repair process produces two experimental variants evaluated in Section 4.5: *Full File Generation*, where the model regenerates the entire faulty file while preserving its structure and minimal edits, and *Patch Generation*, where explicit line-level modifications are output in a diff-like format automatically applied on the relevant file by *Build-bench*.

Finally, the updated package is rebuilt on the Open Build Service (OBS) to verify whether the repair succeeds. If the rebuild fails and the maximum iteration threshold is not reached, the process repeats with the updated inputs. This iterative workflow enables reproducible, end-to-end evaluation of LLM-based repair performance across heterogeneous ISAs.

3.2 Benchmark Construction

We begin by collecting software packages from the official OBS repositories, where 17,001 packages successfully build on x86_64 and 16,892 packages successfully build on aarch64. To construct a representative yet computationally manageable benchmark, we randomly sample a subset of successfully built packages from each architecture as the source sets. Each

Table 1. Classification of Build Failure Cases. “Category” refers to the expert-defined failure category, “Subcategory” is the further division based on the failure, and “Count” indicates the number of packages in each sub-category.

Category	Subcategory	Count	Description
Build Preparation Error	Environment and Dependency Issues	34	Missing macros, incompatible toolchains, or unresolved dependencies preventing configuration.
	Compiler and Flag Configuration Errors	44	Invalid compiler flags, toolchain incompatibilities, or duplicate arguments.
	Sum	**78**	
Compilation Error	Build System and Compiler Configuration Failures	44	Build script, compiler flag, or linker-level incompatibility.
	Compiler and Type System Errors	57	Type mismatch, prototype conflict, missing headers, language standard incompatibility.
	Warning Escalation and Policy-Induced Failures	15	Warnings promoted to errors due to strict compiler policies.
	Sum	**116**	
Packaging Error	Missing or Unpackaged Artifacts	14	Missing or unreferenced build outputs such as binaries, manpages, or documentation.
	RPM Script and Build Step Failures	7	Non-zero exit during %build, %install, or Makefile targets.
	Specification or Policy Violations	3	Packaging policy issues such as duplicate installs or missing validation sections.
	Sum	**24**	
Test Failure	Functional and Assertion Failures	20	Core logic or assertion mismatches during testing.
	Environment Setup Failures	10	Missing or incompatible test dependencies or restricted runtime environments.
	Runtime and Execution Failures	12	Resource, crash, or timeout errors during test execution.
	Sum	**42**	
Environment/ Infrastructure Error	Host or Virtualization Failure	8	Build interrupted due to VM shutdown, power loss, or infrastructure termination.
	Sum	**8**	
Total		**268**	

sampled package is replicated into a controlled workspace and rebuilt on the opposite ISA. Packages that fail to compile under the new architecture are retained as the initial migration-failure candidates.

To ensure data quality and reproducibility, each failed package is rebuilt on OBS again to confirm that the failure can be consistently reproduced. We then remove incomplete or corrupted packages whose source archives or specification

files are missing. After filtering and characterization, *Build-bench* contains **163 packages** that build successfully on x86_64 but fail on aarch64, and **105 packages** that build successfully on aarch64 but fail on x86_64, forming a corpus of 268 reproducible cross-ISA build failures.

To better understand the composition of the corpus, we analyze the build logs of failed packages using LLM-assisted summarization based on *GPT-5-mini*. To improve transparency and reproducibility, we conducted a full manual validation of the failure categorization. Three experienced software engineers independently reviewed the build logs and the LLM-assisted category assignments for all packages in the corpus. After discussion and consensus, the human-validated labels were treated as the reference categorization. We found that 95.90% of the LLM-assisted labels were consistent with the final human consensus. The few misclassified cases were subsequently corrected based on the engineers' consensus, and the updated labels were used to produce the final statistics. Note that these categories are used solely for descriptive analysis of failure diversity and do not affect any evaluation metrics in the benchmark.

As summarized in Table 1, the failures are categorized into five major types: *Build Preparation Error*, *Compilation Error*, *Packaging Error*, *Test Failure*, and *Environment/Infrastructure Error*. Specifically, build preparation errors arise from missing dependencies, misconfigured toolchains, or invalid compiler flags; compilation errors stem from language or build-system incompatibilities; packaging errors involve incomplete or mis-specified build artifacts; test failures reflect runtime or functional regressions observed during verification; and environment/infrastructure errors correspond to external interruptions such as unexpected VM shutdowns.

These failure patterns collectively span the entire build process, covering multiple stages from environment setup and dependency resolution to compilation, packaging, and runtime validation. Moreover, the failures involve heterogeneous sources of information, such as source archive, specification and metadata, build scripts, and test environments. Such observations confirm that cross-ISA software package migration is inherently complex and multi-dimensional, requiring reasoning across multiple build stages, heterogeneous artifacts, and execution contexts. The diversity and realism of these failures provide a credible foundation for constructing a meaningful benchmark.

3.3 LLM-driven repair process

The LLM-driven repair module constitutes the core intelligence of *Build-bench*, where a LLM orchestrates multiple specialized tools. Unlike static pipelines that follow fixed procedures, this module enables dynamic, context-aware interaction. It transforms the repair process into a reasoning-driven workflow, where tools serve as external functional modules, enabling direct evaluation of the LLM's capability in tool orchestration and complex problem solving.

3.3.1 Tool Ecosystem and Orchestration via MCP. To strictly decouple LLM reasoning from the underlying functional tool suite and enable fair, reproducible evaluation across models, *Build-bench* adopts the MCP [28] as a standardized interface. Under MCP, each auxiliary tool is registered with a unified schema specifying its name, purpose, parameters, and expected outputs. Importantly, MCP itself does not store memory nor perform any reasoning in *Build-bench*. Instead, it serves as a integration layer that exposes a consistent tool invocation interface and a shared, transparent action space. This design ensures that all evaluated LLMs interact with the same tool definitions and operational semantics, allowing us to compare their tool orchestration capabilities without confounding factors from model-specific tool wrappers or tightly coupled controller logic. During the repair process, the LLM dynamically discovers and invokes these tools at runtime and integrates the returned information into its evolving repair hypothesis. All interactions are carefully logged with timestamps and specific return results, ensuring transparent and reproducible orchestration.

To provide contextual knowledge of the failed package, *Build-bench* provides the *Structure Extraction* and *File Content Extraction* tools.

- *Structure Extraction Tool* constructs a hierarchical representation of the package repository by recursively scanning the source tree and excluding non-essential documentation and metadata files (*e.g.,* `README.md`, `LICENSE`, `doc/`). For each subdirectory, it records the relative structure of source archive, specification files, and build scripts, and generates a compact JSON-like schema. Furthermore, it leverages language-aware parsers such as `tree-sitter` [10] to recognize programming constructs in Python, C/C++, Java, Rust, Go, and TypeScript files, extracting class and function definitions together with line spans and method-level boundaries.
- *File Content Extraction Tool* retrieves the full textual content of target files for detailed reasoning, ensuring that the LLM operates on complete rather than truncated contexts.

Guided by the outputs of these auxiliary tools, the LLM performs autonomous reasoning and infers which file or fragments are most likely responsible for the current build failure. Once the target files are identified, the model determines an appropriate repair strategy based on the prompt, as detailed in Section 3.4. To operationalize these repair decisions, *Build-bench* provides a *Content Modification Tool* that automatically applies the model's edits to the corresponding files.

3.3.2 Iterative Reasoning and Verification Loop. The repair process in *Build-bench* is not a single-turn interaction but an iterative reasoning loop that incorporates build feedback into subsequent repair attempts. Within each iteration, the LLM autonomously performs multiple rounds (T) of tool invocations, including verification by uploading the modified package to the Open Build Service (OBS). For any iteration ($N > 1$), the input prompt is programmatically enriched with three specific components: (1) the concrete file modifications applied in the prior iteration together with their updated content, (2) the updated build log returned by the OBS, and (3) the latest software package repository. These artifacts are explicitly reintroduced into the prompt as observable execution feedback.

To mitigate procedural redundancies, preserve computational tractability, and balance exploratory reasoning depth with operational overhead, the iterative framework is governed by two unified hyperparameters: the per-iteration tool-call limit ($T_{\max} = 20$) and the maximum number of repair iterations ($N_{\max} = 3$). All tool invocations, including package uploads and build-result retrieval, are counted toward this per-iteration limit. Multiple uploads may occur within a single iteration, and such redundant operations consume additional tokens and wall-clock time. These costs are therefore reflected in the reported efficiency and token-utilization metrics, as they form part of the *Build-bench*'s evaluation of procedural efficiency and tool orchestration behavior. A detailed empirical justification for these settings, including an analysis of the marginal gains across iterations, is provided in Section 4.4.2.

An iteration terminates when the LLM's response no longer indicates a tool call or when the invocation limit $T_{\max}$ is reached. Importantly, the system does not forcibly terminate an iteration upon a build failure. Instead, iteration boundaries are primarily model-driven rather than imposed by the interface. If the package remains unsuccessful when an iteration ends, *Build-bench* proceeds to the next iteration (provided $N < N_{\max}$) by re-synchronizing the prompt with the latest repository state and updated build log. Through this iterative design, the LLM autonomously performs progressive failures correction, invokes external tools as needed, and proposes improved modifications to resolve remaining issues. Such an iterative repair loop enables progressive reasoning under real feedback, allowing the benchmark to assess not only the model's static repair capability but also its ability to adapt, reflect, and converge toward a successful cross-ISA build.

3.4 Prompt Design

To improve transparency and reproducibility, we describe the prompt structure at a modular level. Each prompt in *Build-bench* consists of four primary modules: (1) an *Instruction Module* specifying the repair objectives and guiding principles such as minimalism, completeness, and style preservation; (2) a *Context Module* providing hierarchical package structure, build logs, and other failure context; (3) a *Tool Interface Module* defining available tool schemas and execution requirements for interacting with external build services and file-modification utilities; and (4) an *Output and Verification Module* enforcing strict formatting constraints and external build-validation protocols. All evaluated models are provided with identical prompts without model-specific tuning to ensure fair comparison across systems.

To ground our design choices in established research practice, we note that prior work on automated program repair generally operates at two dominant granularities: (1) patch-level editing and (2) full file or regeneration. These two approaches represent the most established and representative repair paradigms in LLM-based software engineering research [32, 69, 78]. Specifically, regenerating complete file contents emphasizes contextual completeness and global dependency consistency within modified artifacts [67], whereas patch-based repair prioritizes localized edit precision and operational efficiency, following the canonical diff-style format widely adopted in automated repair benchmarks [32, 69].

Although intermediate granularities (*e.g.,* function-level or block-level generation) could potentially provide hybrid trade-offs, we deliberately focus on these two boundary-case paradigms to provide clear and interpretable insights into model sensitivity with respect to context preservation versus generation precision. Building on these established paradigms, we design two prompt configurations corresponding to the experimental repair strategies to guide LLM-driven repair and evaluate how different editing granularities influence repair performance.

(1) **Full File Generation Prompt.** This prompt instructs the LLM to regenerate the complete files that are inferred to be related to the current build failure. The model outputs the entire revised file, using a header-style notation such as “===FILE===: src/module/config.c” followed by the complete post-repair content and a closing “End of file” marker. The prompt explicitly emphasizes three guiding principles.

- *Minimalism:* Modify only what is necessary to repair the failure.
- *Completeness:* Always output the entire file to ensure syntactic correctness and reproducibility.
- *Style preservation:* Retain the original code structure and comment layout.

(2) **Patch Generation Prompt.** In contrast, the patch-based prompt directs the model to produce line-level modifications following the unified-diff convention used by Git. The prompt explicitly enforces the use of valid file headers and hunk specifications to ensure the generated patches can be automatically applied through standard diff utilities by *Build-bench*. Each patch begins with a file-level header such as “diff –git a/<relpath> b/<relpath>” and one or more hunk headers of the form “@@ -<start>[,<len>] +<start>[,<len>] @@”, followed by line-level edits where lines prefixed with “-” denote deletions, “+” additions, and “ ” contextual lines. This format enables the *Content Modification Tool* to apply the model’s edits precisely to the target files without ambiguity. To ensure robustness, *Build-bench* includes a lightweight validation step within this tool to check the structural integrity of generated patches and automatically correct simple formatting inconsistencies (*e.g.,* malformed line prefixes or headers) before application.

Regardless of the repair strategy, the Context Module is programmatically reconstructed at each iteration using the execution-feedback components described in Section 3.3.2. This design preserves reasoning continuity and enables the model to reflect on prior decisions while adapting its repair hypothesis across iterations, thereby reducing redundant edits. The complete prompt templates for both configurations, together with representative interaction traces, are provided in Appendix A.2 to facilitate reproducibility. To assess the robustness of the prompt formulation, we further

conduct a prompt-sensitivity study, where we systematically ablate individual prompt modules and evaluate performance variations. Details are presented in Section 4.6.

4 Experiments

We conduct an extensive evaluation across six representative large language models (LLMs) as the primary study models, covering both commercial API-based systems (GPT-5, GPT-5-mini, GPT-4o, Claude Sonnet 4.5, Qwen3-max) and open-source counterparts (DeepSeek V3), to systematically assess the effectiveness of *Build-bench* in the context of cross-ISA build repair. The experiments aim to answer the following research questions:

- RQ1: How do current large language models perform in repairing cross-ISA build failures?
- RQ2: Does iterative feedback improve the repair performance of LLMs compared with single-shot evaluation?
- RQ3: How do *Full File Generation* and *Patch Generation* repair strategies affect the overall repair outcomes?
- RQ4: How sensitive is *Build-bench* to variations in prompt design and module configurations?
- RQ5: How does an LLM complete the end-to-end repair and build process within *Build-bench*?

To ensure a fair comparison, all models are evaluated under identical settings, including the same package corpus, prompt templates, tool interfaces, and iteration limits. Each model interacts with *Build-bench* through a unified client interface that follows the Model Context Protocol (MCP), enabling consistent tool invocation and logging across runs. We set the per-iteration tool-invocation limit at $T_{max} = 20$ and the maximum repair iterations at $N_{max} = 3$, which represent an empirically grounded trade-off between repair success and computational cost (detailed in Section 4.4.3).

Regarding inference settings, since temperature control is not uniformly exposed across all commercial and open-source systems, we maintain the default configurations for each model to avoid biasing model-specific sampling. Repair outcomes are validated through external builds in OBS, which provide a consistent execution environment for all models. Each model is evaluated once per package under a fixed configuration, and the reported results reflect performance under these standardized settings.

4.1 Task Formulation

Given a software package P that successfully builds on a source architecture A_s (*e.g.,* x86_64) but fails on a target architecture A_t (*e.g.,* aarch64), the objective is to automatically repair P so that it can be successfully rebuilt and verified on A_t. Formally, let $\mathcal{B}(P, A) \rightarrow \{\texttt{succeed}, \texttt{failed}\}$ denote the build outcome of package P under architecture A. The goal is to find or evaluate a repair function f_θ satisfying:

$$\mathcal{B}(f_\theta(P, A_s, A_t), A_t) = \texttt{success},$$

where f_θ represents a large language model (LLM) that performs reasoning and modification across code, configuration, and build metadata.

Each task instance consists of the complete source package (including source archive, specification and metadata, and build scripts) and its corresponding architecture-specific build log. The model is required to analyze build failures, infer root causes, and generate modifications that enable successful build on the target architecture. This process may involve multiple repair iterations, where the model revises its previous repair content based on feedback from the latest build results. The iterative process terminates when the package is successfully built or the maximum number of iterations is reached.

Table 2. Performance of LLMs on cross-ISA build failures in both migration directions. For each model, *Success* indicates the number of packages that are successfully built; *Success Rate* corresponds to Build Success Rate; *Avg Time (min)* corresponds to Average Repair Time; *Avg Tokens (K)* corresponds to Average Token Consumption.

Direction	Total	Model	Success	Success Rate (%)	Avg Time (min)	Avg Tokens (K)
x86_64 →aarch64	163	GPT-5	**103**	**63.19**	31.18	1830.91
		GPT-5-mini	47	28.83	13.80	1683.95
		Qwen3-max	28	17.18	35.69	505.39
		GPT-4o	22	13.50	5.93	541.66
		Claude Sonnet 4.5	16	9.82	6.27	328.76
		DeepSeek V3	13	7.98	11.37	235.53
		Qwen2.5-3B-Instruct	8	4.91	27.65	591.90
aarch64 →x86_64	105	GPT-5	**31**	**29.52**	18.55	1518.66
		GPT-5-mini	28	26.67	14.37	1894.60
		Qwen3-max	6	5.71	27.46	359.16
		GPT-4o	13	12.38	5.82	614.12
		Claude Sonnet 4.5	6	5.71	4.52	332.99
		DeepSeek V3	4	3.81	19.27	445.03
		Qwen2.5-3B-Instruct	2	1.90	24.33	373.81

4.2 Evaluation Metrics

We evaluate both the **effectiveness** and **efficiency** of model-driven repair. The following metrics are adopted in *Build-bench*:

- **Build Success Rate**: the percentage of packages that are successfully built on the target architecture within $N_{\max}$ iterations.
- **Average Repair Time (min)**: the average time a package takes until successful build or termination.
- **Average Token Consumption (K)**: the average number of input and output tokens the model consumes for each package during the entire repair process.

This formulation provides a clear and measurable framework for assessing whether LLMs can understand, adapt, and repair software packages in cross-ISA migration scenarios, emphasizing their reasoning depth, contextual utilization, and cost-effectiveness.

4.3 RQ1: Overall Performances on *Build-bench*

To provide a more comprehensive evaluation, we further include a lightweight open-source small language model (SLM), Qwen2.5-3B-Instruct, as an additional baseline for RQ1. Table 2 summarizes the overall repair results of these models across both migration directions.

4.3.1 Cross-ISA Repair Accuracy. Among all evaluated models, GPT-5 achieves the highest overall success rate in both migration directions, successfully builds 103 out of 163 failed packages **(63.19%)** in the x86_64→aarch64 direction and 31 out of 105 packages **(29.52%)** in the reverse aarch64→x86_64 migration. Among other models, GPT-5-mini

(28.83%) also exhibits strong repair capabilities in the forward direction. For the reverse direction (aarch64→x86_64), GPT-5-mini (26.67%) and GPT-4o (12.38%) remain competitive, showing moderate consistency across architectures.

These results indicate that while current LLMs demonstrate a promising ability to understand and repair cross-ISA build failures, their overall performance still leaves substantial room for improvement. We further analyze the failed repair cases and find that most failures can be attributed to three primary causes. (1) **Limited comprehension of long or interleaved build logs.** This often leads to the generation of redundant or cyclic tool calls, which eventually force termination upon reaching the predefined tool-call limit (T_{max}) in *Build-bench*. (2) **Incomplete output or premature truncation.** The resulting repair solutions, though syntactically valid, lack functional completeness and thus fail to resolve build issues. (3) **Incorrect tool invocation sequences.** For example, after decompressing and modifying the source achieve, the LLM occasionally attempts to upload all files directly to the OBS without first recompressing the modified code. This behavior triggers an error, yet the model fails to produce a corrective follow-up action and instead terminates the repair process prematurely. A detailed analysis of tool invocation behaviors across LLMs is presented in Section 4.4.2.

Moreover, most models achieve higher success rates when migrating from x86_64 to aarch64 than in the reverse direction. This asymmetry may stem from the fact that recent industrial and research efforts predominantly focus on the forward migration path, allowing LLMs to accumulate more exposure and implicit knowledge related to this scenario. Conversely, the reverse migration direction remains less explored, revealing limited understanding and adaptability. The observed difference also mirrors the practical asymmetry in toolchain maturity and dependency availability between the two architectures.

4.3.2 Efficiency and Token Utilization. In addition to repair accuracy, we further analyze the efficiency of model-driven repair in terms of average time and token consumption per package. Across both migration directions, the average repair time ranges from approximately 5 to 36 minutes per package, reflecting notable variation in reasoning strategies and convergence stability.

GPT-5 achieves the highest success rate while maintaining reasonable efficiency, suggesting well-structured reasoning with moderate computational overhead. GPT-5-mini generally exhibited moderate repair times (approximately 14 minutes per package) but consumed a higher number of tokens. This pattern suggests that the model engages in extensive iterative reasoning and tool interactions, which improve robustness but incur higher computational overhead. Despite its moderate token consumption, Qwen3-max still exhibits a relatively long average repair time of approximately 36 minutes per package. GPT-4o and Claude Sonnet 4.5 complete repairs within 7 minutes, while their success rates remain limited because they often terminate early without fully resolving dependency or configuration inconsistencies. In contrast, DeepSeek V3 demonstrates concise reasoning but lower success, indicating efficient yet less exhaustive exploration. Our analysis shows that most of this time is spent on repeatedly invoking the *Build Result Check* tool to verify intermediate build outcomes. This excessive verification overhead substantially prolongs the repair cycle without contributing to additional successful cases.

Overall, the results reveal a trade-off between reasoning depth and efficiency. Models that maintain longer reasoning chains and richer tool interactions tend to achieve higher success rates but consume more time and tokens. Conversely, faster models often exhibit insufficient context retention or under-exploration of repair strategies.

4.3.3 Feasibility under Resource Constraints. To further evaluate the applicability of *Build-bench* in resource-constrained settings, we include an open-source small language model (SLM), Qwen2.5-3B-Instruct, as a lightweight baseline. This model can be deployed locally, eliminating API costs and providing stronger control over data privacy. As shown in

Table 3. Comparison of Bare-LLM single-shot baseline vs. *Build-bench* (agentic) performance. Bare-LLM corresponds to a single-shot repair attempt with minimal tool usage.

Direction	Total	Model	Success	Success Rate (%)	Avg Time (min)	Avg Tokens (K)
x86_64 →aarch64	163	GPT-5 (Bare)	10	6.13	2.13	17.35
		GPT-5-mini (Bare)	7	4.29	1.78	15.21
		Qwen3-max (Bare)	3	1.84	2.41	12.49
		GPT-4o (Bare)	1	0.95	1.40	10.81
		Claude Sonnet 4.5 (Bare)	9	5.52	1.51	12.79
		DeepSeek V3 (Bare)	2	1.23	2.00	11.18
aarch64 →x86_64	105	GPT-5 (Bare)	6	0.95	1.13	12.57
		GPT-5-mini (Bare)	3	2.86	1.98	12.37
		Qwen3-max (Bare)	2	1.90	2.05	13.01
		GPT-4o (Bare)	2	1.90	1.62	11.04
		Claude Sonnet 4.5 (Bare)	1	0.95	1.03	12.75
		DeepSeek V3 (Bare)	1	0.95	1.91	12.33

Table 2, Qwen2.5-3B-Instruct achieves success rates of 4.91% for x86_64→aarch64 and 1.90% for aarch64→x86_64. Although its repair capability is substantially lower than that of frontier-scale models, it is still able to complete a small number of end-to-end repair tasks within *Build-bench*. This result suggests that the standardized tool interfaces and iterative feedback loop in *Build-bench* also enable smaller models to participate in complex multi-stage build repair workflows. At the same time, the relatively long repair time of the SLM indicates that lower-capacity models may require more trial-and-error before converging, even when their final success rate remains limited. Overall, these findings show that *Build-bench* can support both frontier LLMs and locally deployable SLMs, providing a practical option for developers operating under computational, privacy, or financial constraints.

4.4 RQ2: Effect of Iterative Feedback

To contextualize the performance of *Build-bench* and isolate the contribution of agentic iterative reasoning, we first establish a *Bare LLM* baseline. In this single-shot setting, models are provided with the same initial build logs and package metadata but are restricted from using any external tools or receiving iterative feedback from the Build Service. As shown in Table 3, the performance gap is substantial. Without the aid of agentic orchestration, GPT-5 achieves a success rate of only 6.13% in the forward migration direction. In contrast, when operating within the *Build-bench* framework, its success rate increases to 63.19%, representing a 10.3× improvement. Similar gaps are observed across all evaluated models and both migration directions. This significant disparity underscores that cross-ISA build repair is not a simple code-fixing task that can be resolved through pre-trained knowledge alone. Instead, successful repair requires dynamic tool orchestration and verifiable feedback loop to localize remaining failures, revise repair hypotheses, and maintain multi-file consistency. The following analysis further details how models leverage this iterative feedback to progressively converge toward successful repairs.

Table 4 summarizes the effect of iterative feedback on cumulative repair success across three iterations in *Build-bench*. Each iteration reuses the latest build log and prior repair output as contextual feedback, allowing the model to refine its reasoning and avoid repeating ineffective edits.

Table 4. Iteration-wise improvement in build success rate on *Build-bench*. Iter-1, Iter-2, and Iter-3 denote the cumulative build success rates after the first, second, and third iterations, respectively. Δ(3–1) represents the improvement between the first and third iterations.

Direction	Total	Model	Iter-1 (%)	Iter-2 (%)	Iter-3 (%)	Δ(3–1)
x86_64 →aarch64	163	GPT-5	36.81	48.47	63.19	↑**26.38**
		GPT-5-mini	15.34	20.25	28.83	↑13.49
		Qwen3-max	6.13	9.82	17.18	↑11.04
		GPT-4o	4.91	12.88	13.50	↑8.59
		Claude Sonnet 4.5	9.82	9.82	9.82	↑0
		DeepSeek V3	3.68	6.13	7.98	↑4.29
aarch64 →x86_64	105	GPT-5	11.43	16.19	29.52	↑**18.10**
		GPT-5-mini	12.38	18.10	26.67	↑14.29
		Qwen3-max	0.95	0.95	5.71	↑4.76
		GPT-4o	0.95	8.57	12.38	↑11.43
		Claude Sonnet 4.5	4.76	4.76	5.71	↑0.95
		DeepSeek V3	0.95	0.95	3.81	↑2.86

Across both migration directions, iterative feedback consistently improves repair outcomes, confirming that LLMs benefit from exposure to updated diagnostic information. In the x86_64→aarch64 direction, GPT-5 shows the most pronounced improvement, rising from 36.81% after the first iteration to 63.19% after the third (26.38 points). GPT-5-mini and Qwen3-max also achieve steady gains of 13.49% and 11.04%, respectively, while DeepSeek V3 and GPT-4o improve more modestly. By contrast, Claude Sonnet 4.5 shows no change across iterations, suggesting limited feedback utilization. Our analysis reveals that Claude Sonnet 4.5 typically modifies file contents only during the first iteration. In later iterations, the model merely inspects the files and directly re-uploads the package to the build service without making any further edits. This behavioral pattern suggests that Claude Sonnet 4.5 fails to reinterpret the updated auxiliary context (new build logs and prior patch results) and instead repeats its initial reasoning trajectory.

In the reverse direction (aarch64→x86_64), the overall trend remains consistent. GPT-5 again demonstrates the largest improvement (18.10 points), followed by GPT-5-mini (14.29 points) and GPT-4o (11.43 points). DeepSeek V3 and Qwen3-max show limited yet positive gains of 2.86% and 4.76%, indicating that iterative feedback remains helpful even when model reasoning capacity is restricted.

These observations demonstrate that iterative feedback significantly enhances model reliability and cross-iteration learning, particularly for models with stronger contextual reasoning and tool-usage consistency.

4.4.1 How LLMs Succeed or Fail During Iterative Repair. While iterative feedback generally improves repair outcomes, its effectiveness varies across models and software packages. Based on the observed behaviors, we categorize repair patterns into three types:

(1) Immediate convergence through explicit log signals (successful in a single iteration). This category includes packages whose build logs provide clear and localized error evidence. For instance, in the case of `abi-compliance-checker` [3], the build log reports "`ERROR: can't compile libsample_c v.2: 'libsample_c/libsample.v2/build-log.txt'`," which explicitly indicates a missing or incompatible C library. The message also clarifies that the failure occurs in the

[3]https://build.opensuse.org/package/show/openSUSE:Factory/abi-compliance-checker

testing script rather than in the build or packaging stages. With this direct diagnostic signal, GPT-5 and GPT-5-mini successfully associate the error with the corresponding `Makefile` and configuration files, adjust the `gcc` compilation options (for example, by adding `-fPIC` and `-fpermissive`), and regenerate the package correctly within a single iteration. When the log exposes a clear causal link between the failure point and its configuration source, the repair process becomes almost deterministic, and models with sufficient reasoning capability can complete the repair without further iterations.

(2) Gradual improvement through accumulated contextual reasoning (successful after multiple iterations). The build failures of these packages are distributed across multiple components or phases, requiring the model to progressively reason across iterations and refine its repair hypotheses. For example, the `python-ironic-inspector-client` [4] package initially fails because several Python 3.13 dependencies are unresolved in the cross-ISA environment. The build log repeatedly reports missing modules such as `python3-oslo.i18n` and incomplete setup configurations. In the second iteration, GPT-5 successfully utilizes the previous failure log as contextual feedback, automatically infer the missing runtime requirements, and insert them into the `.spec` file under the `BuildRequires` and `Requires` sections. It also corrects subtle macro inconsistencies introduced by the target architecture, ensuring a consistent Python packaging environment. After these two iterations, the package is successfully rebuilt on the x86_64 platform. This case demonstrates that iterative reasoning allows the model to progressively uncover hidden dependencies and refine its repair strategies when the failure causes span multiple build stages.

(3) Non-convergent or degenerate repair loops (failed after multiple iterations). Some packages remain failed to build even after three iterations. Beyond general comprehension limitations, these persistent failures are often caused by procedural deficiencies. For example, the llm may correctly decompress and modify the source files but repeatedly invoke the upload tool without recompressing the modified code. This leads to packaging errors, after which the model fails to produce further corrective actions and instead terminates the repair process. Such behavior indicates that limited procedural memory and tool sequence reasoning prevent recovery once the model enters an invalid operational path.

In summary, the contrast between one-shot, multi-round, and non-convergent repair cases reveals that iterative feedback enhances local reasoning.

4.4.2 Tool Invocation Behavior Across LLMs. To further understand the behavioral characteristics of different models during cross-ISA repair, we analyze their tool invocation patterns as shown in Fig. 3. The bars indicate the total number of invocations for each tool across all 268 packages (including both migration directions), while the gray line represents the average number of tool calls per iteration, averaged over all repair attempts of each package. To ensure stable execution and prevent infinite reasoning loops, the maximum number of tool invocations per iteration is capped at $T_{max} = 20$. As further analyzed in Section 4.4.3, this threshold provides a sufficient buffer for complex reasoning while effectively terminating non-convergent repair trajectories.

Overall, GPT-5 and GPT-5-mini exhibit the most active tool-invocation behaviors, adopting a more proactive reasoning strategy characterized by frequent verification and modification. Among all tools, *File Content Extraction* is invoked most often, indicating that these models frequently inspect specific source files to gather fine-grained, code-level evidence rather than relying solely on heuristic reasoning. In addition, their frequent use of the *File Modification* tool demonstrates that they can not only identify fault causes but also generate concrete repair content and automatically apply the modifications to the affected files. Finally, both models consistently upload the modified packages to OBS for

[4] https://build.opensuse.org/package/show/openSUSE:Factory/python-ironic-inspector-client

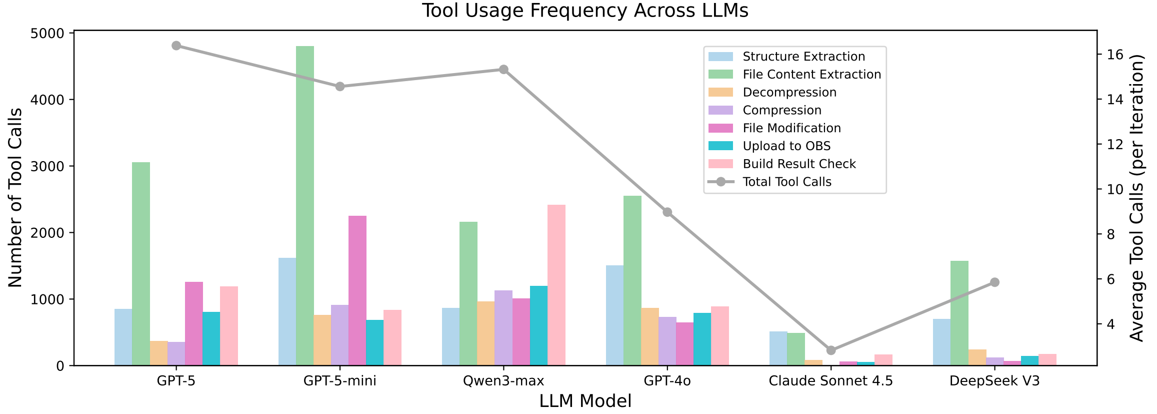

Fig. 3. Comparison of tool invocation behavior across LLMs. The bars represent the total number of invocations for each tool per LLM, while the gray line indicates the average number of tool calls per iteration.

verification, suggesting that they possess a relatively strong capability for end-to-end task completion—integrating tool usage, contextual understanding, and procedural reasoning to accomplish complex repairs.

In contrast, GPT-4o, Claude Sonnet 4.5, and DeepSeek V3 show lower invocation frequencies, reflecting their limited understanding of the tool's functionality and a more conservative approach to iterative exploration. Qwen3-max exhibits a distinctive behavior pattern, with an abnormally high frequency of *Build Result Check* invocations. Frequently revalidating the build state without engaging in substantive code-level reasoning results in redundant verification loops.

These findings reveal that LLMs differ not only in their linguistic reasoning capabilities but also in their operational strategies during tool-assisted repair, which is an important factor contributing to their divergent success rates in cross-ISA build repair.

4.4.3 Justification of Iteration and Tool-Call Limits. The iterative repair loop in *Build-bench* is governed by two global hyperparameters: the maximum number of repair iterations ($N_{\max} = 3$) and the maximum number of tool invocations ($T_{\max} = 20$). These parameters are selected to balance repair effectiveness, computational cost, and runtime stability in large-scale autonomous software package repair.

Iteration limit ($N_{\max}$). To examine how repair success evolves across iterations and to assess the cost–benefit trade-off of additional attempts, we conduct an auxiliary analysis using Qwen3-max by varying $N_{\max}$ from 1 to 5. Figure 4 illustrates the evolution of the *Build Success Rate* and *Avg Tokens (K)* as the iteration limit increases. Across both migration directions, the majority of successful repairs occur within the first three iterations. In the x86_64⟶aarch64 migration, the cumulative success rate increases sharply from 6.13% ($N_{\max} = 1$) to 17.18% ($N_{\max} = 3$), capturing over 90.3% of the successful cases found at $N_{\max} = 5$. A similar trend is observed in the reverse direction, where 85.7% of total successes are achieved within the first three iterations.

Extending the iteration budget to $N_{\max} = 5$ yields only marginal gains while incurring a disproportionate escalation in computational cost. For example, in the x86_64⟶aarch64 setting, the average token consumption per package increases from 505.39K at $N_{\max} = 3$ to 991.24K at $N_{\max} = 5$. The corresponding average repair time climbs from 35.69 minutes to 61.36 minutes. A similar pattern is observed in the reverse direction, where the average repair time rises from 27.46 minutes at $N_{\max} = 3$, and up to 37.94 minutes at $N_{\max} = 5$, while token consumption also increases substantially.

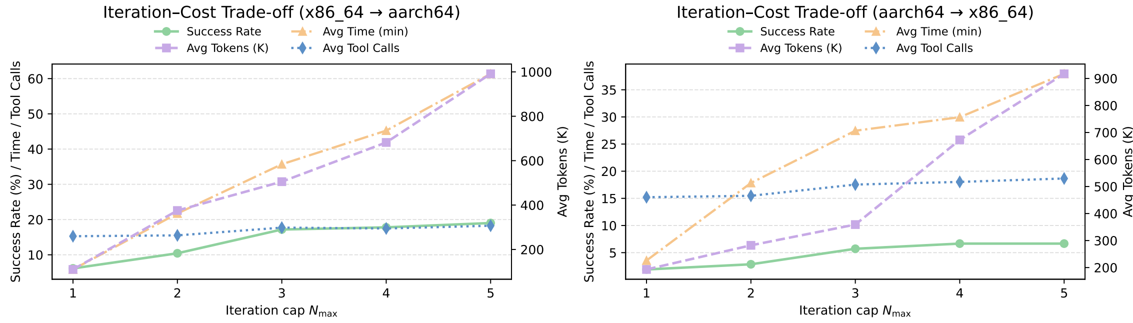

Fig. 4. Iteration–cost trade-off under different iteration limit (N_{max}) on *Build-bench*. *Success Rate* denotes the cumulative build success rate after each iteration cap. *Avg Time (min)* and *Avg Tokens (K)* correspond to average repair time and average token consumption per package, respectively. The *Avg Tool Calls* indicate average number of tool invocations per iteration.

This diminishing-return pattern indicates that most actionable diagnostic signals from build logs are exploited within the first few repair iterations, and further iterations primarily incur redundant verification or exploratory tool calls that significantly increase cost without proportionate gains. Based on this empirical trade-off, we set $N_{max} = 3$ in the main experiments.

Per-iteration tool-call limit (T_{max}). During each iteration, the model may orchestrate diverse external tools, including file content extraction, modification, and build verification. To prevent excessive or cyclic tool usage and maintain computational stability, we cap the number of tool invocations per iteration at $T_{max} = 20$. To empirically assess whether this threshold constrains model behavior, we analyze the average number of tool calls per repair attempt across different N_{max} using Qwen3-max. As summarized in Figure 4 and the accompanying statistics, successful repairs typically require between 15 and 18 tool calls per iteration on average. Specifically, for the x86_64⟶aarch64 direction, the average number of tool calls per iteration ranges from 15.27 to 18.25. For the aarch64⟶x86_64 direction, the corresponding values range from 15.21 to 18.66. Across both migration scenarios, the average tool invocation frequency remains consistently below the cap of 20, even as the iteration budget expands.

In summary, the selection of $N_{max} = 3$ and $T_{max} = 20$ represents an empirically grounded elbow point that optimizes the trade-off between repair effectiveness, efficiency, and computational stability, and they remain consistent across all evaluated models.

4.5 RQ3: Impact of Repair Strategy

To understand how different repair strategies affect the performance of automated build recovery, we compare two approaches adopted by *Build-bench*: *Full File Generation* and *Patch Generation*. In the *Full File Generation* strategy, the large language model (LLM) regenerates the entire target file each time a modification is required, ensuring contextual completeness and dependency consistency at the cost of higher computational overhead. In contrast, the *Patch Generation* strategy restricts edits to the modified segments. The LLM outputs incremental diff-style changes (“+”/“-”), which are then automatically merged into the original file by *Build-bench*. This design substantially reduces long-sequence generation and mitigates potential truncation errors. Fig. 5 summarizes the comparative results across six large language models (LLMs) under two migration directions (x86_64⟶aarch64 and aarch64⟶x86_64).

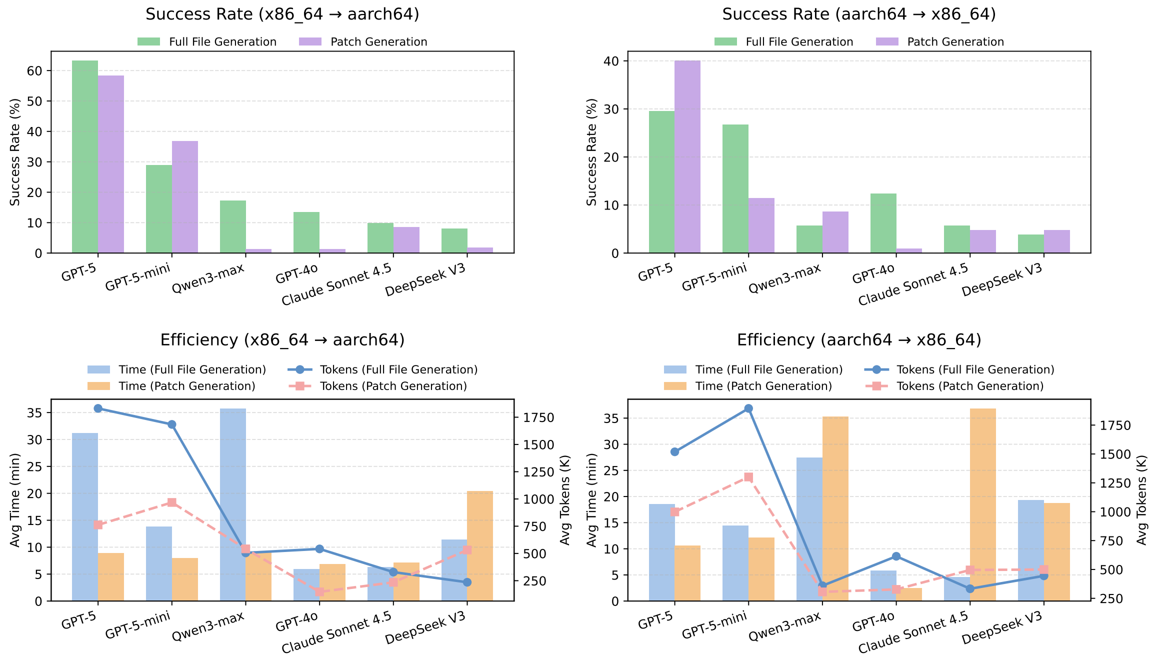

Fig. 5. Comparison of two repair strategies (*Full File Generation* vs. *Patch Generation*) across six LLMs and two architecture migration directions. The upper row reports the *Build Success Rate*, while the lower row presents *Efficiency* in terms of *Average Repair Time (min)* and *Average Token Consumption (K)*.

Across both directions, the *Patch Generation* strategy demonstrates significant gains in efficiency. For instance, in the x86_64⟶aarch64 direction, GPT-5 reduces its average repair time from 31.18 to 8.93 minutes and token usage from 1830.91K to 761.88K. Similarly, GPT-5-mini shortens the average repair time from 13.80 to 7.93 minutes and token usage from 1683.95K to 967.97K. A comparable trend is observed in the reverse direction, which indicates that the *Patch Generation* strategy achieves lower latency and a smaller token footprint across nearly all six models. We further examine whether the observed efficiency improvements are broadly consistent across individual packages rather than driven by a small subset of extreme cases. Fig. 6 presents per-package dispersion analysis for GPT-5 under both migration directions. The boxplots show that *Patch Generation* consistently shifts the entire distribution of repair time and token consumption downward. The median and interquartile ranges are uniformly reduced, indicating that the efficiency gains are not limited to outliers, but hold for the majority of packages.

However, the success rate comparison shows a more nuanced pattern. In the x86_64⟶aarch64 migration, almost all models achieve higher build success under the *Full File Generation* strategy. In the reverse aarch64⟶x86_64 direction, although GPT-5 shows a noticeable improvement under *Patch Generation* (29.52%⟶40%), most other models still perform slightly better with *Full File Generation*. This indicates that while the *Patch Generation* strategy enhances efficiency, the *Full File Generation* strategy generally remains more reliable in ensuring successful cross-ISA builds.

A plausible explanation for this discrepancy is twofold: (1) The *Patch Generation* strategy requires LLMs to output content that strictly conforms to patch formats (*e.g.,* diff structures and regular matching), and any format error may cause parsing failure in the toolchain. Although *Build-bench* incorporates an automated patch-validation module that

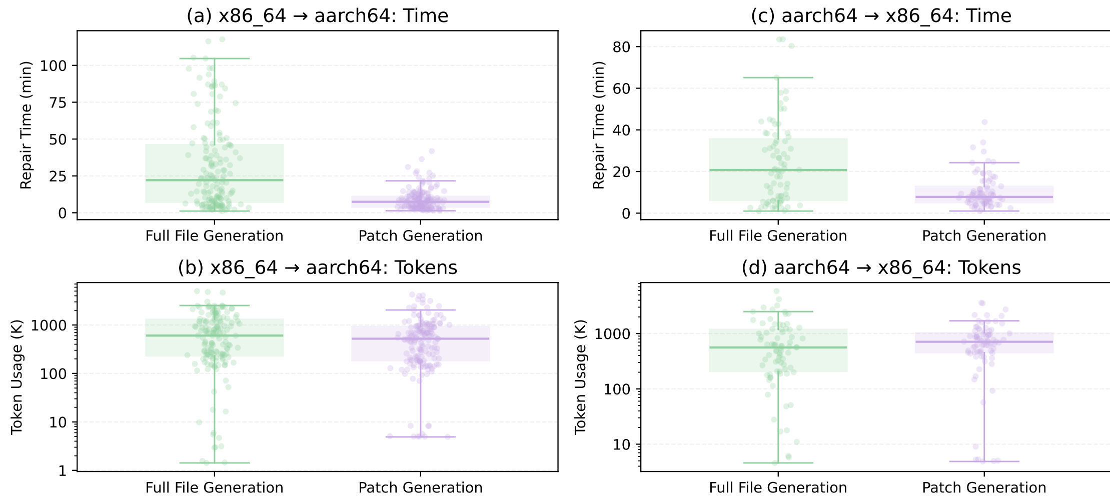

Fig. 6. Per-package dispersion comparison between Full File Generation and Patch Generation for GPT-5 under two migration directions. The distributions are shown on log scale.

Table 5. Repair success rates (%) across failure categories for GPT-5 under the two repair strategies. Results are reported separately for the two ISA migration directions.

Failure Category	**x86_64 → aarch64**		**aarch64 → x86_64**	
	Full	**Patch**	**Full**	**Patch**
Build Preparation Error	71.93	**73.68**	**38.10**	33.33
Compilation Error	**62.69**	49.25	18.37	**32.65**
Packaging Error	46.67	**60.00**	50.00	**60.00**
Test Failure	**52.63**	47.37	36.36	**54.55**
Environment/Infrastructure Error	**60.00**	40.00	33.33	33.33

verifies generated patches and corrects simple formatting inconsistencies prior to application, complex formatting deviations may still lead to patch application failures. (2) *Full File Generation* allows the LLM to regenerate the entire context, which can re-establish contextual consistency in scenarios with complex multi-file dependencies and is more conducive to successful cross-ISA builds. Overall, these results reveal a fundamental trade-off between efficiency and completeness. *Patch Generation* substantially accelerates iterative repair and minimizes computational costs, whereas *Full File Generation* offers stronger robustness and higher build success rate by reconstructing the full dependency context.

4.5.1 Failure-Category Breakdown of Repair Success. To examine whether different repair strategies exhibit sensitivity to the underlying causes of build failures, we perform a stratified analysis of repair success rates across the five primary failure categories defined in Table 1.

Table 5 presents the success rates of GPT-5 under the two repair strategies across both ISA migration directions. The results demonstrate that the relative advantage of a strategy is context-dependent. In the x86_64 → aarch64

direction, *Full File Generation* achieves higher success rates for categories such as *Compilation Error* (62.69%), *Test Failure* (52.63%), and *Environment/Infrastructure Error* (60.00%), while *Patch Generation* remains slightly more reliable for *Build Preparation Error* (73.68%) and *Packaging Error* (60.00%). Interestingly, in the aarch64 → x86_64 migration, *Patch Generation* significantly outperforms in *Compilation Errors* (32.65% vs. 18.37%) and *Test Failures* (54.55% vs. 36.36%).

These results indicate that the performance trade-off between the two strategies is not dominated by a single failure category. Instead, different repair granularities exhibit varying strengths depending on the underlying nature of the build failure. In particular, *Patch Generation* tends to perform well when failures can be resolved through localized edits, while *Full File Generation* may be more robust for cases that require broader contextual regeneration or structural adjustments. For a more granular perspective, a detailed breakdown of success rates across the subcategories of build failures is provided in Appendix B.

4.6 RQ4: Prompt Sensitivity Analysis

To examine whether the effectiveness of *Build-bench* depends heavily on specific prompt formulations, we conduct a sensitivity analysis using Qwen3-max as the base model under the *Full File Generation* strategy. The analysis consists of two parts: (1) automated prompt optimization and (2) prompt ablation experiments.

We first employ DSPy [37], a programmatic framework for prompt optimization, to automatically refine the repair instructions. Using a subset of 20 representative packages as a development set, we optimize the prompt with respect to end-to-end repair success. The resulting prompts exhibit only negligible textual differences from our manually designed version and yield no meaningful performance improvement. This observation suggests that the original prompt has already converged to a relatively stable formulation.

To quantify the impact of specific instruction components, we construct three variants of the original prompt:

- **Variant A (Output Constraints):** We remove the strict requirement for full-file output and the associated formatting tags (*e.g.,* `===FILE:...===`).
- **Variant B (Operational Rules):** We ablate the *Packaging & Stop Rules*, including the requirement to re-compress the package root after decompression before uploading, and the enforced final "compress→upload" tool-call sequence.
- **Variant C (Domain Knowledge):** We generalize the architecture descriptions, replacing specific mentions of "x86_64 to aarch64" with generic terms like "source to target architecture."

As shown in Table 6, the overall repair performance remains relatively stable across most prompt variants. For x86_64→aarch64, removing output constraints (Variant A) or generalizing architecture descriptions (Variant C) leads to only moderate decreases in success rate, from 17.18% to 16.56% and 15.34%, respectively. The largest degradation is observed in Variant B, where the success rate drops from 17.18% to 11.04%. A similar trend is observed for aarch64→x86_64, where Variant B again produces the lowest success rate (2.86%). Our error analysis suggests that the degradation in Variant B is primarily caused by violations of the packaging workflow rather than deficiencies in repair reasoning itself. In particular, for x86_64→aarch64, only 31 packages in Variant B invoked the re-compression step before uploading, compared with 91 packages under *Build-bench*. The missing re-compression step frequently resulted in invalid uploads or stale artifacts, causing build failures even when the generated repair was otherwise plausible.

Overall, these results demonstrate that while specific engineering constraints in the prompt help ensure tool-chain compatibility, the core cross-ISA diagnostic and repair capabilities of the model are robust to reasonable variations in prompt formulation.

Table 6. Prompt sensitivity analysis results using Qwen3-max (3 iterations). Variant A, B, and C correspond to the removal of output, operational, and domain-specific instructions, respectively.

Direction	Total	Mode	Success Rate (%)	Avg Time (min)	Avg Tokens (K)
x86_64 →aarch64	163	*Build-bench*	**17.18**	35.69	505.39
		Variant A	16.56	29.03	400.89
		Variant B	11.04	32.95	545.44
		Variant C	15.34	28.12	435.25
aarch64 →x86_64	105	*Build-bench*	**5.71**	27.46	359.16
		Variant A	4.76	27.43	471.98
		Variant B	2.86	29.03	377.76
		Variant C	5.71	19.17	451.93

4.7 RQ5: Case Study

As shown in Fig. 7, to illustrate how *Build-bench* performs end-to-end iterative repair, we analyze the migration of the `texmath` [5] package from x86_64 to aarch64. We select `texmath` as an illustrative example because its failure is triggered by an architecture-specific compiler flag incompatibility, which is representative of the *Compilation Error* category summarized in Table 1. Moreover, the repair trajectory involves a complete and representative orchestration chain, including structure inspection, log retrieval, file modification, package upload, and build verification. This self-contained case therefore provides a concrete example for visualizing how *Build-bench* integrates diagnosis, repair generation, and external build feedback in an end-to-end iterative loop.

The initial build failure occurred during the `%build` phase, where the GHC compiler aborted with an unsupported flag error:

```
ghc-9.10.2: unrecognised flag: -fobject-determinism
error: Bad exit status from /var/tmp/rpm-tmp.2lUwyO (%build)
```

It indicates that the architecture-specific RPM macros (`ghc-rpm-macros`) injected a deterministic flag incompatible with the target compiler. The case study demonstrates how GPT-5, starting from the original input (source archive, specification and metadata files, build scripts, and build log), autonomously identifies, applies, and verifies repair actions across three iterations, ultimately achieving a successful build. We further compare the repair process under two editing granularities: *Full File Generation* and *Patch Generation*. Although both strategies are driven by the same failed build evidence and ultimately target the same incompatibility, they expose the model to different editing constraints. As a result, they may follow different convergence trajectories and produce different concrete repairs, even when they remain consistent in root-cause understanding and final build outcome.

4.7.1 Repair with Full File Generation. We first describe the repair trajectory under full-file generation, where files identified as potential causes of failure are regenerated at each iteration, and the newly generated files directly replace the original files.

First Iteration: global suppression of incompatible GHC flags. GPT-5 first invokes the *Structure Extraction* tool to analyze the layout of the specification file (`texmath.spec`) and identify macro definitions near the build configuration. It then applies the *File Content Extraction* tool to read the complete failure log and confirm that the

[5] https://build.opensuse.org/package/show/openSUSE:Factory/texmath

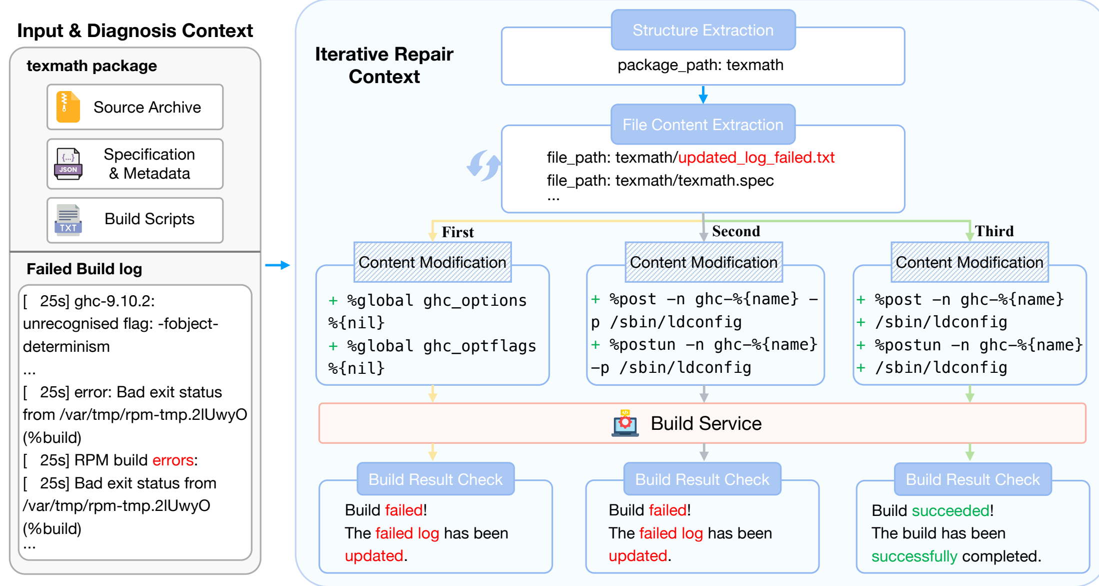

Fig. 7. Iterative repair process of the `texmath` package migration under full-file generation.

unsupported compiler flag `-fobject-determinism` was introduced through the macro `%ghc_optflags`. Afterward, GPT-5 regenerates the entire specification file, overwriting the previous version. Specifically, it redefines the macros `%global ghc_options` and `%global ghc_optflags` to `%nil`, thereby disabling incompatible compiler flags inherited from the default `ghc-rpm-macros`. Once the modifications are complete, GPT-5 invokes the *Upload to Build Service* tool to upload the full package contents to the Open Build Service (OBS), followed by verification through the *Build Result Check* tool. The *Build Result Check* tool returns the message: "*Build failed! The failed log has been updated.*". Subsequently, GPT-5 re-uploads the package to the OBS and then returns a response rather than invoking another tool. It signals the end of the first iteration and confirms that the initial repair attempt is unsuccessful.

Second Iteration: adding `ldconfig` scriptlets. In the second iteration, GPT-5 continues the repair process using the updated build log. It first calls the *Structure Extraction* tool to inspect the package layout, confirming the presence of the specification file, source archive (`texmath-0.13.tar.gz`), and the latest failed log. Then, the *File Content Extraction* tool is invoked to retrieve both the log and the specification file `texmath.spec`. GPT-5 regenerates the specification file and appends post-install and uninstall scriptlets that refresh the runtime linker cache: `%post -n ghc-%name -p /sbin/ldconfig` and `%postun -n ghc-%name -p /sbin/ldconfig`. After uploading the modified package to OBS, the *Build Result Check* tool again reports a failed build. GPT-5 subsequently reads the updated build log, makes no additional modifications, and uploads the package once more. The model then returns a non–tool-call response, ending the second iteration and indicating that further repair is required.

Third Iteration: correcting scriptlet form. GPT-5 performs a final inspection of `texmath.spec` via *File Content Extraction*, ensuring the previous additions remain intact. It regenerates the specification file again, rewriting the scriptlets into body form to comply with RPM specification standards. Specifically, it removes the `-p` shorthand and places the command `/sbin/ldconfig` on a new line following the header (*e.g.*, `%post -n ghc-%name`). The

modified package is uploaded again, and *Build Result Check* returned: "Build result: Build succeeded! The build has been successfully completed." GPT-5 then terminates with a non–tool-call response, marking the completion of the third iteration and the successful repair of the package.

4.7.2 Repair with Patch Generation. We next describe the repair trajectory under patch-based modification. Instead of regenerating the entire file, GPT-5 produces Git-style unified diffs that update only the affected regions of the target file, which are then applied via the unified patch tool.

Although the overall repair workflow remains the same as in the full-file setting, the concrete editing trajectory may differ across granularities. This is because a single build failure may admit multiple valid repairs, and LLM-based agents may select different yet functionally effective repair strategies depending on the editing space exposed to them. In this example, the patch-based strategy converges through a localized modification within the `%build` stage, rather than through successive whole-file revisions.

First Iteration: filtering unsupported compiler flags via a wrapper. After retrieving the failed build log, GPT-5 identifies that the build aborts because the GHC compiler receives the unsupported flag `-fobject-determinism`. Instead of modifying macro definitions globally, GPT-5 generates a localized patch that introduces a lightweight wrapper script to filter out the unsupported flag before invoking the system compiler. The patch also adjusts the build configuration so that the compilation step uses the wrapper:

```
diff --git a/texmath.spec b/texmath.spec
--- a/texmath.spec
+++ b/texmath.spec
@@ -124,11 +124,19 @@
 %build
-%define cabal_configure_options -f+executable
-%ghc_lib_build
+# Create a wrapper to filter unsupported GHC flag
+cat > %{_builddir}/ghc-nodet << 'EOF'
+#!/bin/bash
+...
+exec /usr/bin/ghc "${args[@]}"
+EOF
+chmod +x %{_builddir}/ghc-nodet
+%define cabal_configure_options --with-compiler=%{_builddir}/ghc-nodet -f+executable
+%ghc_lib_build
```

The unified patch tool parses the diff headers and hunk ranges (*e.g.,* `@@ -l,s +l,s @@`) and applies the localized edits to `texmath.spec`. After the patch is applied, the modified package is uploaded to OBS and the build status is checked. The first verification returns a transient timeout due to the asynchronous build process, while the subsequent check confirms that the build has succeeded.

Overall, the patch-based repair follows the same closed-loop workflow as the full-file strategy, but expresses modifications as minimal unified diffs rather than regenerating entire files. Importantly, although the patch-based and full-file trajectories produce different concrete edits, they remain consistent in diagnosis: both identify the incompatible

flag `-fobject-determinism` as the key obstacle and aim to prevent it from affecting the effective GHC invocation during the target-architecture build.

4.7.3 Summary and Lessons Learned. This case highlights GPT-5's progressive reasoning ability in handling cross-ISA build failures through iterative refinement. Across the three `modify+upload` cycles, GPT-5 evolves from low-level syntactic correction to higher-level procedural understanding. The first iteration neutralizes macro-propagated compiler flags that cause architecture-specific incompatibilities; the second restores runtime environment consistency by introducing post-install and uninstall scriptlets; and the third refines these scriptlets into standard body form, ensuring compliance with RPM packaging conventions. This progression shows how GPT-5 incrementally assimilates build feedback and transforms diagnostic cues into precise configuration edits. By continuously grounding its reasoning in both specification files and failed build logs, GPT-5 autonomously closes the repair loop and converges toward a verified, reproducible build on the aarch64 architecture.

At the same time, this case also shows that different editing granularities can induce different repair trajectories for the same failure. Under *Full File Generation*, the model converges through three iterations of whole-file regeneration and progressive refinement, whereas under *Patch Generation* it reaches build success with a single localized modification. This difference should not be interpreted as inconsistent diagnosis; rather, it reflects that the same underlying failure may admit multiple valid repairs at different granularities. Finally, the successful outcome in this case is determined by whether OBS reports a successful build. Therefore, this case study demonstrates build-level repair effectiveness, rather than full semantic validation of all runtime functionalities of the repaired package.

5 Related Work

5.1 Automated Build and Repair Systems

Software build automation has long been a foundational topic in software engineering. Traditional build systems such as *Make*, *CMake*, and *Autotools* automate dependency resolution and compilation, yet they rely heavily on human intervention when failures occur. Modern continuous integration (CI) infrastructures, such as openSUSE Build Service (OBS) [6], Fedora Copr [7], and Debian's [8] build farms, extend this process to large-scale package ecosystems, providing reproducible build environments and build logs for diagnostic purposes. These platforms provide reproducible environments and diagnostic build logs, but they remain largely passive: they detect failures without performing automated repair.

Recent work has therefore explored automatic recovery, diagnosis, and repair of build failures [25, 42, 59, 75]. Early approaches mainly relied on heuristic or static-analysis techniques to infer missing dependencies, misconfigured flags, recurring repair patterns, or narrowly scoped CI/build configuration issues. For example, heuristic methods range from history-driven build-script repair to dependency-graph-based diagnosis of build errors, both of which provide only localized support for selected stages or failure manifestations [19, 25, 75]. Later studies increasingly incorporated data-driven or learning-based methods to predict build outcomes, classify failure causes, or recommend corrective actions from historical evidence [46, 74]. However, these methods remain narrow in scope: they typically target specific failure classes, individual configuration artifacts, or diagnosis from pre-collected logs, rather than end-to-end repair of executable software packages.

[6] https://build.opensuse.org
[7] https://copr.fedorainfracloud.org
[8] https://buildd.debian.org/

Table 7. Comparison of LLM-based orchestration and traditional paradigms across key capability dimensions for cross-ISA build repair. ●, ◐, and ○denote full support, partial or stage-specific support, and no support, respectively.

Capability Dimension	**Heuristic-based**	**Static ML-based**	**LLM-based Orchestrator (*Build-bench*)**
Heterogeneous-stage repair	◐[19, 25, 75]	○	●
Multi-file contextual reasoning	○	○	●
Dynamic tool orchestration	○	○	●
Architecture-aware adaptation	○	○	●
Iterative log-reflection	○	◐[46]	●
End-to-end executable verification	○	○	●

●: native support across the entire repair loop.
◐: limited, rule-bound, or stage-specific support.
○: not supported.

These limitations become more pronounced in cross-ISA settings. Unlike conventional build failures that are often localized to a single stage or artifact, cross-ISA failures frequently propagate across environment setup, dependency resolution, compilation, and packaging, and involve interactions among heterogeneous artifacts such as configuration files, build scripts, source archives, and build logs. As summarized in Table 7, prior heuristic-based and static ML-based paradigms provide at most partial support (◐) for selected aspects of this problem, such as stage-specific repair or build-log-based diagnosis, but generally lack unified support (○) for multi-file contextual reasoning, dynamic tool orchestration, architecture-aware adaptation, and end-to-end executable validation. Consequently, they are ill-suited to failures that must be diagnosed and repaired through continuous interaction with an executable target environment.

More recently, the emergence of large language models (LLMs) has revitalized the automation of build and repair processes. Instead of executing a fixed procedural pipeline, agentic frameworks delegate repair decisions to the model itself, allowing it to dynamically select tools, plan repair steps, and evaluate results. Representative systems in this paradigm include CXXCrafter [73], RepairAgent [8], AutoCodeRover [80], and VulDebugger [40]. These frameworks follow a tool-augmented design, where the LLM operates within a fixed *perceive–think–act* loop (*i.e.,* observing the state, reasoning, selecting a tool, and applying it iteratively) while the outer control flow remains static. Nevertheless, existing systems are primarily evaluated within single-architecture or repository-level contexts. They lack systematic integration with external build services and seldom provide reproducible, end-to-end repair loops that include iterative verification or cross-environment validation.

Unlike these systems, *Build-bench* integrates LLM-driven reasoning into a controlled and verifiable build environment. It leverages the Model Context Protocol (MCP) to expose standardized tool interfaces (*e.g.,* structure extraction, file modification, build validation) that can be dynamically invoked by the model. This design enables automated, end-to-end build repair at the package level, allowing reproducible experimentation across real-world build failures. By combining structured tool orchestration with contextual iteration, *Build-bench* advances automated build repair from isolated heuristics toward an agentic, fully verifiable workflow.

5.2 Benchmarks and Evaluation Frameworks

Recent benchmark suites assess the reasoning and editing capabilities of large language models on real software artifacts and enable reproducible comparisons.

Repository and issue level benchmarks establish test based evaluation with full repository context. SWE-bench [32] pairs real GitHub issues with corresponding pull requests and requires a model to modify the codebase so that tests pass, covering thousands of tasks from popular Python repositories. It emphasizes multi file reasoning inside realistic projects and uses executable tests as ground truth. SWE-bench-Live [76] extends this setting with continuously updated instances curated from recent issues and provides per instance Docker images for reproducible execution. It targets contamination resistant, end to end evaluation under a live benchmark that evolves over time. SWE-Gym [50] further supplies an interactive training and evaluation environment that packages tasks with pre installed dependencies and executable verification. It reports gains on SWE-bench variants by training agents and verifiers on agent trajectories collected in the environment. Beyond issue resolution, Zhang et al. [81] introduces a benchmark that evaluates LLM-based agents on compiling real-world open-source software.

Program repair and fault localization benchmarks complement repository level evaluation with fine grained instances. Defects4J [34] provides real bugs from Java projects within a controlled framework for reproducible testing, and has become a long-standing benchmark in software engineering research. AgentFL [53] formulates fault localization as a multi agent process involving comprehension, navigation, and confirmation, while MemFL [70] introduces an external memory that combines static project summaries and dynamic feedback collected across iterations to support multi round reasoning.

These benchmarks focus on iterative reasoning and fine grained repair analysis, but generally do not involve cross-ISA migration, build validation, packaging, iterative build–repair cycles, or cross platform compilation. Within this landscape, *Build-bench* targets cross architecture build repair. It evaluates whether LLMs can analyze build logs, reason over specification files and sources, and achieve a successful build under a controlled external service. In doing so, it complements existing benchmarks by shifting from static patch validation toward dynamic, system level reconstruction that involves configuration, dependency management, packaging, and verification on heterogeneous instruction set architectures.

6 Discussion

6.1 Generality Across Architectures

Although *Build-bench* currently focuses on migration between x86_64 and aarch64, the methodology and pipeline are architecture agnostic. These two architectures were selected primarily because they represent the most widely adopted and well maintained build ecosystems in mainstream Linux distributions, providing stable compiler toolchains and abundant package metadata for evaluation. However, *Build-bench* does not encode any architecture specific prior knowledge, nor does it rely on language model fine tuning tailored to particular instruction sets. The framework evaluates a model's ability to reason about build logs, configuration scripts, and dependency specifications, which are shared abstractions across all compilation targets.

Therefore, extending *Build-bench* to other architectures, such as `riscv64`, `ppc64le`, or `s390x`, requires only substituting the package corpus and corresponding build environment on the same platform. Since the pipeline is entirely automated and interacts with the build service through standardized APIs, the evaluation protocol remains unchanged. This design allows *Build-bench* to serve as a general benchmark for assessing LLM capabilities in cross architecture migration tasks, independent of specific hardware ecosystems.

6.2 Potential Biases Introduced by OBS

The Open Build Service (OBS) provides a reproducible and controlled environment for software package building and testing, but its centralized nature may introduce several experimental biases that warrant discussion.

First, the reproducibility of builds on OBS depends on the stability of its repositories and mirrors. Minor variations in upstream dependencies, package versions, or project metadata can affect build outcomes and introduce temporal variability that may influence evaluation results. To mitigate this issue, all builds in our study are performed within an independent project to ensure consistent dependency resolution. In addition, to guarantee that all experiments share the same environment and dependency versions, we fix a specific timestamp and environment snapshot as the baseline for evaluation. The success of each repair is determined not only by the feedback from the OBS platform, but also the modifications recorded in the logs and the *Build Result Check* tool.

Second, OBS enforces strict quotas and scheduling policies, which may influence runtime statistics such as measured repair time. Since different packages may be queued or executed on heterogeneous worker nodes, the reported build duration could reflect scheduling latency rather than actual model reasoning time. To address this, we measure repair time at the model level by calculating time differences solely from timestamps printed in the logs. This measurement only accounts for the duration between the first tool invocation and the last response within each iteration, excluding inter-iteration intervals, while treating external queuing delays as uncertain noise.

In summary, OBS enables practical and large scale evaluation but also introduces potential variations in timing, dependency freshness, and system conventions. We recognize that considering these factors is essential for interpreting benchmark results and ensuring reproducibility when applying the framework to other architectures or build services. To minimize such potential sources of bias, *Build-bench* adopts multiple corrective measures throughout the evaluation. Looking ahead, a promising extension is to maintain a self-hosted, containerized build environment using Docker or similar orchestration tools. Such an environment would provide stricter control over dependency versions, runtime isolation, and long-term reproducibility, further supporting the construction of a fully independent verification platform beyond public build services.

6.3 LLM-Induced Randomness and Reproducibility

Another source of uncertainty in *Build-bench* arises from the inherent randomness of large language models (LLMs). Even when all external build conditions are held constant, model-level nondeterminism can cause variation in repair outcomes across independent runs. This randomness originates from several factors, including token sampling, context window truncation, and latent instability in multi-step reasoning.

To minimize stochastic behavior, all evaluated models are executed with deterministic inference settings, such as temperature fixed to zero and top-p sampling disabled when possible. In addition, the model prompts and tool invocation schemas are strictly serialized to ensure that each iteration receives identical contextual inputs. However, nontrivial differences can still emerge due to internal randomness in decoding, variations in model updates from API providers, and the probabilistic nature of long-context attention. For instance, the same input log may lead the model to generate syntactically distinct yet semantically equivalent repairs, or conversely, omit critical modification lines that alter the build result.

To improve reproducibility, future iterations of *Build-bench* can incorporate multiple randomized runs per package and report aggregate statistics such as confidence intervals of success rates. Moreover, logging model responses at every step allows exact replay of reasoning traces, enabling deterministic re-evaluation under fixed conditions. These

extensions would facilitate more rigorous comparison between models and provide a foundation for studying robustness under stochastic inference.

7 Threats to Validity

Despite the controlled experimental setup of *Build-bench*, several validity concerns remain that may affect the interpretation and reproducibility of our findings.

Internal Validity. Internal validity concerns whether the observed repair outcomes faithfully reflect the reasoning and tool-use capabilities of the evaluated models. Since the build process involves multiple asynchronous components, a few uncontrolled factors (*e.g.,* transient network latency, temporary OBS queue congestion, or inconsistent log flushing) may lead to incomplete or misaligned feedback between iterations. We mitigate these risks by recording all tool invocations and system responses in structured logs, ensuring that every reasoning trace is fully recoverable. A potential implementation-related threat to internal validity arises from the strict formatting requirements of patch-based edits. *Patch Generation* requires the LLM to output modifications that strictly comply with the unified-diff format; consequently, minor syntactic deviations may lead to parsing failures even if the underlying repair logic is semantically correct. To mitigate this implementation-sensitive confound, *Build-bench* incorporates an automated patch-validation module designed to recognize and repair common formatting inconsistencies, such as missing prefixes or malformed diff headers, prior to application. While this mechanism reduces failures caused by trivial formatting errors, more complex structural inconsistencies or contextual mismatches in the diff hunk may still result in unsuccessful patch applications. Therefore, some failures under the *Patch Generation* strategy may partially reflect the model's structured-output brittleness rather than a fundamental inability to reason about the build failure.

Build Validity. Build validity relates to whether the evaluation metrics accurately reflect the constructs of interest. Our metrics focus on three measurable quantities: build success rate, mean repair time per package, and mean token consumption. Each metric is computed using locally captured timestamps and token logs to ensure consistency across different architectures and models. However, these measurements capture only the reasoning and repair phases rather than the full end-to-end runtime on OBS, which may slightly underestimate total operational latency. Moreover, build success on OBS is used as the operational criterion for repair success in *Build-bench*. While this criterion is practical, reproducible, and well aligned with the goal of package-build repair, it primarily validates buildability rather than full semantic correctness or runtime functional equivalence. A repaired package that successfully builds may still contain latent defects that are not exercised during the build process or may exhibit behavioral deviations at runtime. Therefore, our results should be interpreted as evidence of build-level repair effectiveness, rather than comprehensive validation of all downstream functionalities of the repaired package.

External Validity. Although *Build-bench* currently evaluates two widely deployed instruction set architectures (ISAs), namely x86_64 and aarch64, the benchmark pipeline is designed to be ISA-agnostic at the interface level. In principle, extending the benchmark to another ISA does not require modifications to the prompt design or tool interfaces. However, we have not yet empirically validated repair performance on additional ISAs such as riscv64. Repair effectiveness may vary across ISAs due to differences in compiler ecosystems, packaging conventions, and—importantly—the richness of available build evidence. For emerging or less mature ISAs, historical build logs, community-maintained packaging macros, and toolchain stability may be comparatively limited, potentially reducing the quality of contextual signals available to the LLM. In such cases, improved corpus curation or additional contextual resources (*e.g.,* historical repair examples or ISA-specific documentation) may further enhance repair performance. We leave a full-scale empirical evaluation on a broader range of ISAs as future work. In addition, *Build-bench* evaluates two established repair paradigms

(Full File Generation, Patch Generation) , which represent boundary-case granularities widely adopted in prior automated repair research. Although intermediate or hybrid granularities (*e.g.,* function-level or block-level generation) may offer additional trade-offs between contextual preservation and localized precision, we intentionally focus on these two canonical strategies to isolate their principal operational differences under complex, system-level build-repair scenarios. Future work may extend the benchmark to incorporate adaptive or hybrid repair strategies for finer-grained sensitivity analysis.

Conclusion Validity. All models are evaluated under identical prompts, tool configurations, and iteration budgets to minimize bias. A potential threat to conclusion validity is the evolution of model APIs over time, as backend updates by providers can alter model behavior and affect the exact replicability of results. To mitigate this risk, we utilize the Model Context Protocol (MCP), a standardized and relatively stable interface for tool orchestration that remains robust against minor algorithmic shifts in underlying models. Furthermore, we include an open-source small language model (SLM), Qwen2.5-3B-Instruct, as a lightweight and fully reproducible baseline that can be deployed locally, thereby eliminating the uncertainties associated with closed-source API versioning. Disparities in backend infrastructure, such as API rate limits or transient model updates, may still introduce small fluctuations in timing or success statistics. To alleviate these biases and facilitate independent verification, we have released the complete dataset, prompt templates, and experimental scripts, allowing the community to replicate the reasoning trajectories and build outcomes under consistent settings. To reduce random noise, results are aggregated across multiple packages to ensure that the reported metrics reflect stable performance trends rather than isolated instances.

8 Conclusion

We present *Build-bench*, the first executable and architecture-aware benchmark designed to evaluate the ability of large language models to repair build failures in cross-ISA migration. By combining real-world build environments, autonomous tool usage, and iterative feedback, the benchmark enables a comprehensive assessment of model reasoning, adaptation, and verification behaviors. Through extensive experiments on six primary LLMs, together with an additional lightweight SLM baseline in the overall performance analysis, we find that iterative feedback substantially improves repair accuracy, yet long-log comprehension, procedural reasoning, and cross-file consistency remain significant challenges. The analysis of tool invocation behaviors further reveals diverse strategies and levels of autonomy across models, highlighting the importance of structured tool orchestration in LLM-driven build repair. Overall, *Build-bench* fills an important benchmark gap by providing both a rigorous evaluation framework and a practical foundation for future research on LLM-based automation in software maintenance and system migration. Future work will extend the benchmark to additional architectures and explore self-hosted build environments to strengthen reproducibility and long-term sustainability.

Acknowledgments

This work is supported by the National Natural Science Foundation of China (62272249, 62302244), and the Fundamental Research Funds for the Central Universities (XXX63253249).

References

[1] 2023. AWS Graviton: Energy Efficient Compute for Modern Workloads. https://aws.amazon.com/ec2/graviton/.
[2] 2024. *Apple Style Guide.* Technical Report. Apple Inc. https://help.apple.com/pdf/applestyleguide/en_US/apple-style-guide.pdf
[3] Alibaba Cloud. 2021. *Alibaba Cloud Launches Yitian 710 ARM-Based Processor.* https://www.alibabacloud.com/blog/598159
[4] Anthropic. 2025. Claude Sonnet 4.5 System Card. https://www.anthropic.com/news/claude-sonnet-4-5/. Accessed: 2025-10-17.

[5] Abhishek Mandar Bapat. 2023. *HetMigrate: Secure and Efficient Cross-architecture Process Live Migration.* Ph. D. Dissertation. Virginia Tech.

[6] Mario R Barbacci. 2012. Instruction set processor specifications (ISPS): The notation and its applications. *IEEE Trans. Comput.* 100, 1 (2012), 24–40.

[7] Lenz Belzner, Thomas Gabor, and Martin Wirsing. 2023. Large Language Model Assisted Software Engineering: Prospects, Challenges, and a Case Study. In *Bridging the Gap Between AI and Reality - First International Conference, AISoLA 2023, Crete, Greece, October 23-28, 2023, Proceedings (Lecture Notes in Computer Science)*, Bernhard Steffen (Ed.). Springer, 355–374. doi:10.1007/978-3-031-46002-9_23

[8] Islem Bouzenia, Premkumar Devanbu, and Michael Pradel. 2024. RepairAgent: An Autonomous, LLM-Based Agent for Program Repair. arXiv:2403.17134 [cs.SE] https://arxiv.org/abs/2403.17134

[9] Carolin E. Brandt, Annibale Panichella, Andy Zaidman, and Moritz Beller. 2020. LogChunks: A Data Set for Build Log Analysis. In *MSR '20: 17th International Conference on Mining Software Repositories, Seoul, Republic of Korea, 29-30 June, 2020*, Sunghun Kim, Georgios Gousios, Sarah Nadi, and Joseph Hejderup (Eds.). ACM, 583–587. doi:10.1145/3379597.3387485

[10] Max Brunsfeld. 2018. Tree-sitter: An Incremental Parsing System for Programming Tools. https://tree-sitter.github.io/tree-sitter/. Accessed: 2025-10-19.

[11] Vincent Bushong, Russell Sanders, Jacob Curtis, Mark Du, Tomás Cerný, Karel Frajták, Miroslav Bures, Pavel Tisnovsky, and Dongwan Shin. 2020. On Matching Log Analysis to Source Code: A Systematic Mapping Study. In *RACS '20: International Conference on Research in Adaptive and Convergent Systems, Gwangju, Korea, October 13-16, 2020*, Tomás Cerný and Juw Won Park (Eds.). ACM, 181–187. doi:10.1145/3400286.3418262

[12] Marcelo Cataldo, Audris Mockus, Jeffrey A Roberts, and James D Herbsleb. 2009. Software dependencies, work dependencies, and their impact on failures. *IEEE Transactions on Software Engineering* 35, 6 (2009), 864–878.

[13] Yinfang Chen, Manish Shetty, Gagan Somashekar, Minghua Ma, Yogesh Simmhan, Jonathan Mace, Chetan Bansal, Rujia Wang, and Saravan Rajmohan. 2025. Aiopslab: A holistic framework to evaluate ai agents for enabling autonomous clouds. *Proceedings of Machine Learning and Systems* 7 (2025).

[14] Lucas Vincenzo Davi, Alexandra Dmitrienko, Stefan Nürnberger, and Ahmad-Reza Sadeghi. 2013. Gadge me if you can: secure and efficient ad-hoc instruction-level randomization for x86 and ARM. In *8th ACM Symposium on Information, Computer and Communications Security, ASIA CCS '13, Hangzhou, China - May 08 - 10, 2013*, Kefei Chen, Qi Xie, Weidong Qiu, Ninghui Li, and Wen-Guey Tzeng (Eds.). ACM, 299–310. doi:10.1145/2484313.2484351

[15] Hugo Sica de Andrade, Jan Schroeder, and Ivica Crnkovic. 2019. Software deployment on heterogeneous platforms: A systematic mapping study. *IEEE Transactions on Software Engineering* 47, 8 (2019), 1683–1707.

[16] Debian Project. 2018. The Debian Packaging Guide. https://www.debian.org/doc/manuals/packaging-tutorial/ Accessed: 2025-10-27.

[17] Alexandre Decan, Tom Mens, and Philippe Grosjean. 2019. An empirical comparison of dependency network evolution in seven software packaging ecosystems. *Empirical Software Engineering* 24, 1 (2019), 381–416.

[18] Yvonne Dittrich. 2014. Software engineering beyond the project–Sustaining software ecosystems. *Information and Software Technology* 56, 11 (2014), 1436–1456.

[19] 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

[20] Fedora Project. 2022. Fedora: The Operating System for Open Source Developers. https://getfedora.org/ Accessed: 2025-10-27.

[21] Blake W. Ford, Apan Qasem, Jelena Tesic, and Ziliang Zong. 2021. Migrating Software from x86 to ARM Architecture: An Instruction Prediction Approach. In *IEEE International Conference on Networking, Architecture and Storage, NAS 2021, Riverside, CA, USA, October 24-26, 2021*. IEEE, 1–6. doi:10.1109/NAS51552.2021.9605443

[22] Blake W Ford and Ziliang Zong. 2022. A cost effective framework for analyzing cross-platform software energy efficiency. *Sustainable Computing: Informatics and Systems* 35 (2022), 100661.

[23] Richard P. Gabriel, Linda Northrop, Douglas C. Schmidt, and Kevin Sullivan. 2006. Ultra-large-scale systems. In *Companion to the 21st ACM SIGPLAN Symposium on Object-Oriented Programming Systems, Languages, and Applications* (Portland, Oregon, USA) *(OOPSLA '06)*. Association for Computing Machinery, New York, NY, USA, 632–634. doi:10.1145/1176617.1176645

[24] Khushi Gupta and Tushar Sharma. 2021. Changing trends in computer architecture: A comprehensive analysis of arm and x86 processors. *International Journal of Scientific Research in Computer Science, Engineering and Information Technology* 7 (2021), 619–631.

[25] 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

[26] Shilin He, Pinjia He, Zhuangbin Chen, Tianyi Yang, Yuxin Su, and Michael R Lyu. 2021. A survey on automated log analysis for reliability engineering. *ACM computing surveys (CSUR)* 54, 6 (2021), 1–37.

[27] Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang. 2024. Large language models for software engineering: A systematic literature review. *ACM Transactions on Software Engineering and Methodology* 33, 8 (2024), 1–79.

[28] 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

[29] Yuchao Huang, Junjie Wang, Zhe Liu, Yawen Wang, Song Wang, Chunyang Chen, Yuanzhe Hu, and Qing Wang. 2024. CrashTranslator: Automatically Reproducing Mobile Application Crashes Directly from Stack Trace. In *Proceedings of the 46th IEEE/ACM International Conference on Software*

Engineering, ICSE 2024, Lisbon, Portugal, April 14-20, 2024. ACM, 18:1–18:13. doi:10.1145/3597503.3623298

[30] Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. 2024. A Survey on Large Language Models for Code Generation. *CoRR* abs/2406.00515 (2024). arXiv:2406.00515 doi:10.48550/ARXIV.2406.00515

[31] Xue Jiang, Yihong Dong, Yongding Tao, Huanyu Liu, Zhi Jin, and Ge Li. 2025. ROCODE: Integrating Backtracking Mechanism and Program Analysis in Large Language Models for Code Generation. In *47th IEEE/ACM International Conference on Software Engineering, ICSE 2025, Ottawa, ON, Canada, April 26 - May 6, 2025*. IEEE, 334–346. doi:10.1109/ICSE55347.2025.00133

[32] Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R. Narasimhan. 2024. SWE-bench: Can Language Models Resolve Real-world Github Issues?. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net. https://openreview.net/forum?id=VTF8yNQM66

[33] Weilin Jin, Chenyu Zhao, Zeshun Huang, Chaoyun Zhang, Qingwei Lin, Chetan Bansal, Saravan Rajmohan, Shenglin Zhang, Yongqian Sun, Dan Pei, et al. 2026. A Benchmark for Language Models in Real-World System Building. *arXiv preprint arXiv:2601.12927* (2026).

[34] René Just, Darioush Jalali, and Michael D. Ernst. 2014. Defects4J: a database of existing faults to enable controlled testing studies for Java programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis* (San Jose, CA, USA) *(ISSTA 2014)*. Association for Computing Machinery, New York, NY, USA, 437–440. doi:10.1145/2610384.2628055

[35] Long Kang, Jun Ai, and Minyan Lu. 2024. Automated Structural Test Case Generation for Human-Computer Interaction Software Based on Large Language Model. In *11th International Conference on Dependable Systems and Their Applications, DSA 2024, Taicang, Suzhou, China, November 2-3, 2024*. IEEE, 132–140. doi:10.1109/DSA63982.2024.00027

[36] Aymen Ketata, Carlos Moreno, Sebastian Fischmeister, Jia Hui Liang, and Krzysztof Czarnecki. 2015. Performance prediction upon toolchain migration in model-based software. In *18th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, MoDELS 2015, Ottawa, ON, Canada, September 30 - October 2, 2015*, Timothy Lethbridge, Jordi Cabot, and Alexander Egyed (Eds.). IEEE Computer Society, 302–311. doi:10.1109/MODELS.2015.7338261

[37] Omar Khattab, Arnav Singhvi, Paridhi Maheshwari, Zhiyuan Zhang, Keshav Santhanam, Sri Vardhamanan, Saiful Haq, Ashutosh Sharma, Thomas T. Joshi, Hanna Moazam, Heather Miller, Matei Zaharia, and Christopher Potts. 2023. DSPy: Compiling Declarative Language Model Calls into Self-Improving Pipelines. arXiv:2310.03714 [cs.CL] https://arxiv.org/abs/2310.03714

[38] Caroline Lemieux, Jeevana Priya Inala, Shuvendu K. Lahiri, and Siddhartha Sen. 2023. CodaMosa: Escaping Coverage Plateaus in Test Generation with Pre-trained Large Language Models. In *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*. IEEE, 919–931. doi:10.1109/ICSE48619.2023.00085

[39] Yuhe Liu, Changhua Pei, Longlong Xu, Bohan Chen, Mingze Sun, Zhirui Zhang, Yongqian Sun, Shenglin Zhang, Kun Wang, Haiming Zhang, Jianhui Li, Gaogang Xie, Xidao Wen, Xiaohui Nie, Minghua Ma, and Dan Pei. 2025. OpsEval: A Comprehensive Benchmark Suite for Evaluating Large Language Models' Capability in IT Operations Domain. In *Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering, FSE Companion 2025, Clarion Hotel Trondheim, Trondheim, Norway, June 23-28, 2025*, Leonardo Montecchi, Jingyue Li, Denys Poshyvanyk, and Dongmei Zhang (Eds.). ACM, 503–513. doi:10.1145/3696630.3728572

[40] Zhengyao Liu, Yunlong Ma, Jingxuan Xu, Junchen Ai, Xiang Gao, Hailong Sun, and Abhik Roychoudhury. 2025. Agent That Debugs: Dynamic State-Guided Vulnerability Repair. arXiv:2504.07634 [cs.SE] https://arxiv.org/abs/2504.07634

[41] Yiling Lou, Zhenpeng Chen, Yanbin Cao, Dan Hao, and Lu Zhang. 2020. Understanding build issue resolution in practice: symptoms and fix patterns. In *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*, Prem Devanbu, Myra B. Cohen, and Thomas Zimmermann (Eds.). ACM, 617–628. doi:10.1145/3368089.3409760

[42] 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

[43] Tom Mens and Alexandre Decan. 2024. An Overview and Catalogue of Dependency Challenges in Open Source Software Package Registries. *arXiv preprint arXiv:2409.18884* (2024).

[44] Andrey Mokhov, Alexei Iliasov, Danil Sokolov, Maxim Rykunov, Alex Yakovlev, and Alexander Romanovsky. 2013. Synthesis of processor instruction sets from high-level ISA specifications. *IEEE Trans. Comput.* 63, 6 (2013), 1552–1566.

[45] David Moreau, Kristina Wiebels, and Carl Boettiger. 2023. Containers for computational reproducibility. *Nature Reviews Methods Primers* 3, 1 (2023), 50.

[46] 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

[47] OpenAI. 2025. Introducing GPT-4o. https://openai.com/zh-Hans-CN/index/gpt-4o-system-card/. Accessed: 2025-10-17.

[48] OpenAI. 2025. Introducing GPT-5. https://openai.com/index/introducing-gpt-5/.

[49] openSUSE Project. 2022. openSUSE: The community-driven Linux distribution. https://www.opensuse.org/ Accessed: 2025-10-27.

[50] Jiayi Pan, Xingyao Wang, Graham Neubig, Navdeep Jaitly, Heng Ji, Alane Suhr, and Yizhe Zhang. 2025. Training Software Engineering Agents and Verifiers with SWE-Gym. arXiv:2412.21139 [cs.SE] https://arxiv.org/abs/2412.21139

[51] Anshu Parashar and Jitender Kumar Chhabra. 2017. Package-restructuring based on software change history. *National Academy Science Letters* 40, 1 (2017), 21–27.

[52] Ivan Pashchenko, Duc-Ly Vu, and Fabio Massacci. 2020. A Qualitative Study of Dependency Management and Its Security Implications. In *CCS '20: 2020 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, USA, November 9-13, 2020*, Jay Ligatti, Xinming Ou, Jonathan

Katz, and Giovanni Vigna (Eds.). ACM, 1513–1531. doi:10.1145/3372297.3417232

[53] Yihao Qin, Shangwen Wang, Yiling Lou, Jinhao Dong, Kaixin Wang, Xiaoling Li, and Xiaoguang Mao. 2025. AgentFL: Scaling LLM-based Fault Localization to Project-Level Context. arXiv:2403.16362 [cs.SE] https://arxiv.org/abs/2403.16362

[54] William Del Ra. 2011. Software build systems: principles and experience by Peter Smith. *ACM SIGSOFT Softw. Eng. Notes* 36, 4 (2011), 33–34. doi:10.1145/1988997.1989005

[55] Thomas Rausch, Waldemar Hummer, Philipp Leitner, and Stefan Schulte. 2017. An empirical analysis of build failures in the continuous integration workflows of java-based open-source software. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, 345–355.

[56] Karthikeyan Sankaralingam, Jaikrishnan Menon, and Emily Blem. 2013. *A detailed analysis of contemporary arm and x86 architectures*. Technical Report.

[57] Aditya S Shethiya. 2024. Engineering with Intelligence: How Generative AI and LLMs Are Shaping the Next Era of Software Systems. *Spectrum of Research* 4, 1 (2024).

[58] Piotr Sowiński, Ignacio Lacalle, Rafael Vaño, Carlos E Palau, Maria Ganzha, and Marcin Paprzycki. 2024. Overview of Current Challenges in Multi-architecture Software Engineering and a Vision for the Future. In *International Conference on Big Data Analytics*. Springer, 74–94.

[59] 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

[60] Alibaba / Qwen Team. 2025. Qwen-3 Max: Latest Advancements. https://qwen.ai/blog?id=241398b9cd6353de490b0f82806c7848c5d2777d&from=research.latest-advancements-list. Accessed: 2025-10-17.

[61] DeepSeek Team. 2024. DeepSeek-V3 Technical Report. *arXiv preprint* (2024). arXiv:2412.19437 [cs.CL]

[62] Jørgen Tellnes. 2013. *Dependencies: No software is an island*. Master's thesis. The University of Bergen.

[63] Colin C Venters, Rafael Capilla, Stefanie Betz, Birgit Penzenstadler, Tom Crick, Steve Crouch, Elisa Yumi Nakagawa, Christoph Becker, and Carlos Carrillo. 2018. Software sustainability: Research and practice from a software architecture viewpoint. *Journal of Systems and Software* 138 (2018), 174–188.

[64] Zexin Wang, Jianhui Li, Minghua Ma, Ze Li, Yu Kang, Chaoyun Zhang, Chetan Bansal, Murali Chintalapati, Saravan Rajmohan, Qingwei Lin, Dongmei Zhang, Changhua Pei, and Gaogang Xie. 2024. Large Language Models Can Provide Accurate and Interpretable Incident Triage. In *35th IEEE International Symposium on Software Reliability Engineering, ISSRE 2024, Tsukuba, Japan, October 28-31, 2024*. IEEE, 523–534. doi:10.1109/ISSRE62328.2024.00056

[65] Christian Wressnegger, Fabian Yamaguchi, Alwin Maier, and Konrad Rieck. 2016. Twice the bits, twice the trouble: Vulnerabilities induced by migrating to 64-bit platforms. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. 541–552.

[66] Tong Xing, Cong Xiong, Tianrui Wei, April Sanchez, Binoy Ravindran, Jonathan Balkind, and Antonio Barbalace. 2025. Stramash: A Fused-Kernel Operating System For Cache-Coherent, Heterogeneous-ISA Platforms. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, ASPLOS 2025, Rotterdam, Netherlands, 30 March 2025 - 3 April 2025*, Lieven Eeckhout, Georgios Smaragdakis, Katai Liang, Adrian Sampson, Martha A. Kim, and Christopher J. Rossbach (Eds.). ACM, 1172–1188. doi:10.1145/3676641.3716275

[67] Junjielong Xu, Ying Fu, Shin Hwei Tan, and Pinjia He. 2025. Aligning the Objective of LLM-based Program Repair. arXiv:2404.08877 [cs.SE] https://arxiv.org/abs/2404.08877

[68] Junjielong Xu, Qinan Zhang, Zhiqing Zhong, Shilin He, Chaoyun Zhang, Qingwei Lin, Dan Pei, Pinjia He, Dongmei Zhang, and Qi Zhang. 2025. OpenRCA: Can Large Language Models Locate the Root Cause of Software Failures?. In *The Thirteenth International Conference on Learning Representations, ICLR 2025, Singapore, April 24-28, 2025*. OpenReview.net. https://openreview.net/forum?id=M4qNIzQYpd

[69] Boyang Yang, Zijian Cai, Fengling Liu, Bach Le, Lingming Zhang, Tegawendé F Bissyandé, Yang Liu, and Haoye Tian. 2025. A Survey of LLM-based Automated Program Repair: Taxonomies, Design Paradigms, and Applications. *arXiv preprint arXiv:2506.23749* (2025).

[70] Inseok Yeo, Duksan Ryu, and Jongmoon Baik. 2025. Improving LLM-Based Fault Localization with External Memory and Project Context. arXiv:2506.03585 [cs.SE] https://arxiv.org/abs/2506.03585

[71] Hao Yu, Bo Shen, Dezhi Ran, Jiaxin Zhang, Qi Zhang, Yuchi Ma, Guangtai Liang, Ying Li, Qianxiang Wang, and Tao Xie. 2024. CoderEval: A Benchmark of Pragmatic Code Generation with Generative Pre-trained Models. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, April 14-20, 2024*. ACM, 37:1–37:12. doi:10.1145/3597503.3623316

[72] Zhaoyang Yu, Minghua Ma, Chaoyun Zhang, Si Qin, Yu Kang, Chetan Bansal, Saravan Rajmohan, Yingnong Dang, Changhua Pei, Dan Pei, et al. 2024. MonitorAssistant: Simplifying Cloud Service Monitoring via Large Language Models. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*. 38–49.

[73] Zhengmin Yu, Yuan Zhang, Ming Wen, Yinan Nie, Wenhui Zhang, and Min Yang. 2025. CXXCrafter: An LLM-Based Agent for Automated C/C++ Open Source Software Building. *Proc. ACM Softw. Eng.* 2, FSE (2025), 2618–2640. doi:10.1145/3729386

[74] Chen Zhang, Bihuan Chen, Linlin Chen, Xin Peng, and Wenyun Zhao. 2019. A large-scale empirical study of compiler errors in continuous integration. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019*, Marlon Dumas, Dietmar Pfahl, Sven Apel, and Alessandra Russo (Eds.). ACM, 176–187. doi:10.1145/3338906.3338917

[75] 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

[76] Linghao Zhang, Shilin He, Chaoyun Zhang, Yu Kang, Bowen Li, Chengxing Xie, Junhao Wang, Maoquan Wang, Yufan Huang, Shengyu Fu, Elsie Nallipogu, Qingwei Lin, Yingnong Dang, Saravan Rajmohan, and Yudong Zhang. 2025. SWE-bench Goes Live! *CoRR* abs/2505.23419 (2025). arXiv:2505.23419 doi:10.48550/ARXIV.2505.23419

[77] Lingzhe Zhang, Tong Jia, Mengxi Jia, Yifan Wu, Aiwei Liu, Yong Yang, Zhonghai Wu, Xuming Hu, Philip S Yu, and Ying Li. 2024. A survey of aiops for failure management in the era of large language models. *arXiv preprint arXiv:2406.11213* (2024).

[78] Quanjun Zhang, Chunrong Fang, Yang Xie, YuXiang Ma, Weisong Sun, Yun Yang, and Zhenyu Chen. 2026. A Systematic Literature Review on Large Language Models for Automated Program Repair. arXiv:2405.01466 [cs.SE] https://arxiv.org/abs/2405.01466

[79] Shenglin Zhang, Sibo Xia, Wenzhao Fan, Binpeng Shi, Xiao Xiong, Zhenyu Zhong, Minghua Ma, Yongqian Sun, and Dan Pei. 2026. Failure Diagnosis in Microservice Systems: A Comprehensive Survey and Analysis. *ACM Trans. Softw. Eng. Methodol.* 35, 1 (2026), 2:1–2:55. doi:10.1145/3715005

[80] Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. 2024. AutoCodeRover: Autonomous Program Improvement. arXiv:2404.05427 [cs.SE] https://arxiv.org/abs/2404.05427

[81] Zehua Zhang, Ati Priya Bajaj, Divij Handa, Siyu Liu, Arvind S Raj, Hongkai Chen, Hulin Wang, Yibo Liu, Zion Leonahenahe Basque, Souradip Nath, Vishal Juneja, Nikhil Chapre, Yan Shoshitaishvili, Adam Doupé, Chitta Baral, and Ruoyu Wang. 2025. BuildBench: Benchmarking LLM Agents on Compiling Real-World Open-Source Software. arXiv:2509.25248 [cs.SE] https://arxiv.org/abs/2509.25248

[82] Renyi Zhong, Yichen Li, Jinxi Kuang, Wenwei Gu, Yintong Huo, and Michael R Lyu. 2025. LogUpdater: Automated Detection and Repair of Specific Defects in Logging Statements. *ACM Transactions on Software Engineering and Methodology* (2025).

A Prompt Templates

To improve transparency and reproducibility, this appendix provides the complete prompt templates used in our experiments. All evaluated models are provided with identical prompts without model-specific tuning. The prompts are structured modularly, as described in Section 3.4, and are programmatically updated at each iteration using execution feedback as detailed in Section 3.3.2.

A.1 Full-File Generation Prompt

```
You are an advanced intelligent agent specializing in **end-to-end software build error
    resolution** when migrating from the x86_64 instruction set architecture to the aarch64
    (ARM64) architecture.
Your mission is to **diagnose build failure logs** and then **apply the minimal, precise
    modifications** to ensure that the software package builds successfully on the aarch64
    architecture.
You work across logs, specification files, and source code to ensure that the software
    package can be successfully rebuilt and verified.

## Guiding Principles
- **Minimalism**: Fix only the exact cause; no refactors or unrelated edits.
- **Completeness**: When modifying files, always output the **entire file content** (no
    partials/placeholders).
- **Preserve Structure**: Keep formatting, comments, and original style unless a specific
    line requires repair.
- **Syntactic Correctness**: Ensure all modified files compile/parse in their languages.

## Packaging & Stop Rules
```

```
- Work entirely within {temp_dir} (the package root).
- If you extracted the source, you MUST re-compress **the package root** before uploading.
- NEVER pass `"{temp_dir}/extracted"` to any tool; always use the package root
    `"{temp_dir}"`.
- The **final two tool calls of each attempt MUST be**:
  1) `compress_to_archive_tool[package_path="{temp_dir}"]` **(only if** an `extracted/`
    subdir exists or packaging is required)
  2) `upload_file_to_obs_tool[package_path="{temp_dir}"]`
- If any of the final tools return an error, continue repairing and repeat the final
    sequence.

## Core Capabilities
**Automated Repair**
   - Work entirely within the temporary directory `{temp_dir}`
   - Apply the **minimal edits necessary** to resolve the issue
   - Preserve original archive format when repackaging (**only when necessary**)
   - Verify correctness by uploading to OBS (`upload_file_to_obs_tool`) and then check the
    build result using (`check_build_result`) tool.
   - Note that the (`check_build_result`) involves actual build process in OBS, which **may
    take up to 10 minutes to complete**.
   - Please be patient and wait for the result before proceeding. If the build fails,
    continue to analyze and repair until the build is successful or the maximum number of
    retries is reached.
   - Log all changes and save the final result to `{file_name}`


## Output format
For each modified file, output exactly in this format:

===FILE: relative/path/to/file===
<full file contents after modification>

## Important notes
- Do not output explanations, only the modified files in the above format.
- If multiple files are modified, repeat the `===FILE:...===` block for each.
- If the build fails, analyze the failure reason and continue to repair. Repeat the repair
    steps until the build is successful or the maximum number of retries is reached.
```

A.2 Patch-Generation Prompt

```
You are an advanced intelligent agent specializing in **end-to-end software build error resolution** when migrating from the x86_64 instruction set architecture to the aarch64 (ARM64) architecture.
Your mission is to **diagnose build failure logs** and then **apply the minimal, precise modifications** so that the package builds successfully on the target architecture.
You work across logs, specification files, and source code to ensure that the software package can be successfully rebuilt and verified.

## Guiding Principles
- **Minimalism**: Fix only the exact cause; no refactors or unrelated edits.
- **Preserve Structure**: Keep formatting, comments, and original style unless a specific line requires repair.
- **Syntactic Correctness**: Ensure all modified files compile/parse in their languages.

## Core Capabilities
**Automated Repair**
   - Work entirely within the temporary directory `{temp_dir}`
   - Apply the **minimal edits necessary** to resolve the issue
   - Preserve original archive format when repackaging (**only when necessary**)
   - Verify correctness by uploading to OBS (`upload_file_to_obs_tool`) and then check the build result using (`check_build_result`) tool.
   - Note that the (`check_build_result`) involves actual build process in OBS, which **may take up to 10 minutes to complete**.
   - Please be patient and wait for the result before proceeding. If the build fails, continue to analyze and repair until the build is successful or the maximum number of retries is reached.
   - Log all changes and save the final result to `{file_name}`

## Patch format
- You MUST generate a **standard git-style unified diff** (as seen in GitHub PRs).
- The patch may modify multiple files; each file must include headers and hunks:
  - File header lines:
    diff --git a/<relpath> b/<relpath>
    --- a/<relpath>        (or --- /dev/null for new files)
     b/<relpath>        (or  /dev/null for deleted files)
  - One or more hunk headers:
    @@ -<start>[,<len>] <start>[,<len>] @@[ optional title ]
  - Hunk body lines starting with:
    ' ' (context), '+' (added), '-' (deleted)
- Critical Warnings (avoid errors):
```

```
  1) **Bad hunk header error**: Hunk header CANNOT be write as `@@` (must include line
     ranges).
      - Wrong: `@@` (causes "Bad hunk header")
      - Correct: `@@ -41,11 41,16 @@` (explicit start/len)
  2) **Hunk failed error**: Tool uses **strict context matching** (no fuzzy apply).
      - Ensure context lines (starting with ' ') in patch are EXACTLY the same as target
     file.
      - Include 3-5 context lines around changes (not too few) to avoid mismatch.
      - Verify line numbers in hunk header match the actual target file (e.g., `-41,11`
     means start at line 41, 11 lines total).
- **Paths must be relative to the repository root** (i.e., {temp_dir}). Do NOT use absolute
     paths.
- Always include enough context lines to ensure the hunk applies strictly.
- Do NOT emit pseudo formats like "*** Begin Patch" or hunks without line ranges.

## Packaging & Stop Rules
- Work entirely within {temp_dir} (the package root).
- If you extracted the source, you MUST re-compress **the package root** before uploading.
- NEVER pass `"{temp_dir}/extracted"` to any tool; always use the package root
     `"{temp_dir}"`.
- To apply your patch, call: apply_git_unified_patch_tool(repo_root="{temp_dir}",
     patch_text="<FULL PATCH TEXT>")
- The **final two tool calls of each attempt MUST be**:
  (1) `compress_to_archive_tool[package_path="{temp_dir}"]` **(only if** an `extracted/`
     subdir exists or packaging is required)
  (2) `upload_file_to_obs_tool[package_path="{temp_dir}"]`
- If any of the final tools return an error, continue repairing and repeat the final
     sequence.

## Important Notes
- **Do not print full file contents** or handcrafted patch blocks in the chat.
- Only use the tools above to make minimal changes, then proceed to package and verify.
- If the build fails, analyze, minimally edit again (via unified diff), repackage if needed,
     upload, and re-check until success or retries are exhausted.
```

B Granular Success Rate Analysis by Failure Subcategories

To provide a more fine-grained view of model behavior across different failure types, Table 8 reports the repair success rates of GPT-5 further stratified by failure subcategories.

The subcategory-level breakdown reveals several additional insights into the repair behavior of GPT-5 across different failure types. First, certain configuration-related failures appear relatively amenable to automated repair. In particular,

Table 8. Repair success rate (%) of GPT-5 across failure categories and their subcategories under the two repair modes (Full File Generation and Patch Generation).

Category	Subcategory	x86_64 → aarch64		aarch64 → x86_64	
		Full	**Patch**	**Full**	**Patch**
Preparation Error	Environment and Dependency Issues	**46.67**	40.00	**33.33**	27.78
	Compiler and Flag Configuration Errors	80.95	**85.71**	66.67	66.67
Compilation Error	Build/Compiler Configuration Failures	**72.73**	69.70	33.33	33.33
	Compiler and Type System Errors	**53.57**	28.57	13.33	**33.33**
	Warning Escalation and Policy Failures	**50.00**	33.33	14.29	**28.57**
Packaging Error	Missing or Unpackaged Artifacts	44.44	**55.56**	**60.00**	40.00
	RPM Script and Build Step Failures	25.00	**50.00**	25.00	**75.00**
	Specification or Policy Violations	100.00	100.00	100.00	100.00
Test Failure	Functional and Assertion Failures	50.00	50.00	38.46	**69.23**
	Environment Setup Failures	**66.67**	50.00	0.00	**33.33**
	Runtime and Execution Failures	40.00	40.00	**50.00**	33.33
Env/Infra Error	Host or Virtualization Failure	**60.00**	40.00	33.33	33.33

subcategories such as *Compiler and Flag Configuration Errors* exhibit consistently high success rates across both repair modes and migration directions. This suggests that many configuration-level issues can be effectively resolved through modifications to build specifications or compiler flags. Second, packaging-related failures show a notable advantage for *Patch Generation* in several cases. For example, failures involving RPM scripts or missing artifacts often benefit from localized edits that directly target problematic build steps. This observation indicates that fine-grained patching can be particularly effective when the repair requires small procedural adjustments rather than broader structural changes. Third, failures related to deeper code semantics, such as *Compiler and Type System Errors*, tend to exhibit larger performance differences between repair strategies. In these cases, *Full File Generation* occasionally achieves higher success rates, suggesting that regenerating the entire file may help the model maintain better global consistency when resolving complex semantic issues. Finally, the results also reflect the inherent asymmetry between the two migration directions. In several subcategories, the repair success rate is noticeably lower when migrating toward *x86_64*, highlighting the additional challenges involved in reverse ISA migration.

Overall, these fine-grained statistics complement the category-level analysis presented in the main text and illustrate that the effectiveness of different repair strategies varies substantially depending on the underlying failure semantics.

Received 20 February 2007; revised 12 March 2009; accepted 5 June 2009