COMBENCH: A REPOSITORY-LEVEL REAL-WORLD BENCHMARK FOR COMPILATION ERROR REPAIR

Anonymous authors
Paper under double-blind review

ABSTRACT

Compilation errors frequently interrupt software development, yet existing Automated Compilation Error Repair (ACER) benchmarks poorly reflect real-world practice because they rely on decontextualized single-file tasks and narrow data sources. We present COMBENCH, the first repository-level, reproducible real-world benchmark for C/C++ compilation error repair. COMBENCH mines failures from GitHub Actions histories of large open-source projects, identifies developer repair patches from version histories, and reconstructs historical build environments with Docker. Each instance is execution-verified to reproduce the original failure and pass after applying the ground-truth fix. The released benchmark contains 200 errors from OpenSSL, Bitcoin, RocksDB, and LLVM, covering seven error categories with full repository context, compiler diagnostics, and developer patches. Using COMBENCH, we evaluate 12 modern LLMs under direct and agent-based repair settings. Results show a large gap between compilation success and semantic correctness: GPT-5 achieves 73% compile success, but only 41% semantic correctness. We also find model-specific strengths across error types and settings. COMBENCH offers a realistic testbed for developing ACER methods that handle repository-level software complexity.

1 INTRODUCTION

Compilation errors (com, 2025) are a pervasive and disruptive challenge within the core "edit-compile-debug" software development cycle. According to a large-scale study (Seo et al., 2014), C++ developers experience compilation failures in 37.4% of their 10.1 daily builds, while Java developers face a 29.7% failure rate across 6.98 daily builds. Resolving these errors manually is a time-consuming and tedious task, with reported median repair times of 5 mins for C++ and 12 mins for Java, exhibiting an order of magnitude variation across error types. This forces programmers to switch from feature development to troubleshooting, leading to blockages and delays that impact overall productivity.

Automated compilation error repair (ACER) techniques (Gupta et al., 2017; Mesbah et al., 2019; Li et al., 2023; Wu et al., 2024; Deligiannis et al., 2025) are proposed to alleviate these issues. The advent of LLMs has significantly enhanced the effectiveness of code generation, which boosts the development of LLM-based ACER tools (Deligiannis et al., 2025). However, fully assessing their performance is challenging due to a lack of benchmarks reflecting real-world compilation error complexity. Existing ACER benchmarks (Gupta et al., 2017; Ahmed et al., 2018) deviate significantly from the practices of real-world software development across three dimensions: *data granularity*, *source authenticity*, and *task modeling*. First, their decontextualized, single-file paradigm (typically around 20 lines) isolates errors from their project environment, preventing the capture of essential repository-level complexities, such as cross-file references and inter-module dependencies. Second, their narrow data sources have led to a lack of authenticity and diversity. Existing benchmarks are exclusively derived from introductory students' programming assignments. This results in a relatively homogeneous set of error types (*e.g.*, missing delimiters). Thus, they fail to represent the complex and diverse errors in large-scale projects. Lastly, this oversimplified data setting leads to biased task models that frame ACER as a local, sequence-to-sequence problem. It overlooks critical challenges such as global fault localization and comprehensive context collection within large repositories. Consequently, existing benchmarks create a significant disconnect between academic research and

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

industrial practice. This highlights the urgent need for a more comprehensive and representative benchmark.

To address these limitations, we propose COMBENCH, a repository-level real-world benchmark for compilation error repair with reproducible environments and developer-provided ground-truth fixes. COMBENCH aims to provide a unified platform for more accurately evaluating the performance of ACER methods. It features several distinct advantages. (1) *Repository-level Context:* COMBENCH's data is directly sourced from Git repository histories of open-source projects. Each error instance includes the complete project-level context, *i.e.*, all project code, file structure, and dependencies. This enables repair methods to consider cross-file context. (2) *Authentic Real-world Errors*: Compilation errors are collected from GitHub's Continuous Integration (CI) system (ciw, 2025), GitHub Actions (github, 2025a). Error-inducing code and its corresponding repair patches are derived from actual developer commits, ensuring the authenticity and representativeness of the data. (3) *Comprehensive Error Types with Labels*: The benchmark encompasses a wide variety of compilation error types, from simple syntax to complex cross-file issues. Each error instance includes detailed compiler diagnostics, complete code context, and the original developer's repair patch, which is essential for evaluating repair intent and correctness. (4) *Reproducible and Continuously Evolving*: COMBENCH provides Docker containers for consistent compilation environments, ensuring reproducibility of reported errors. Furthermore, a reusable data collection pipeline ensures continuous updates, allowing the benchmark to evolve with new errors and development practices from GitHub.

The construction of such a benchmark presents three significant challenges. First, *the heterogeneity and noise in CI logs* make it difficult to precisely extract structured compiler diagnostics. Second, a more complex challenge lies in *the causal inference of the ground truth repair*, specifically locating the exact fix commit and isolating its minimal repair patch from multi-purpose changes. Finally, the critical challenge is *reproduction*: accurately recreating the exact historical CI environment, complicated by inaccessible proprietary runner images and ambiguous job configurations. A detailed discussion is provided in Appendix A.

Our COMBENCH framework systematically addresses these challenges through a targeted, multi-faceted approach. To identify compilation errors from noisy CI logs, we employ a dual-filtering algorithm complemented by a bidirectional context collector. For causally inferring the ground truth repair, COMBENCH utilizes a multi-source search algorithm followed by compilation dependency analysis to isolate the minimal repair patch. Finally, we ensure reproducibility through a two-layer Docker-based environment emulation and an intelligent matching algorithm for job configuration inference. Details are provided in Appendix A.

To ensure the correctness of collected data, we perform end-to-end reproducible validation for each candidate instance: we reproduce the compilation failure in the reconstructed environment and verify that applying the developer fix makes the build succeed (i.e., fail-fix-pass). We retain only instances that pass this check; all 200 released instances satisfy it.

Our constructed benchmark, COMBENCH, comprises 200 C/C++ compilation errors, encompassing seven major categories of compilation issues. These include prevalent declaration, type, member, and syntax errors, as well as less frequent function, template, and semantic errors. These errors are sourced from four prominent large-scale C/C++ open-source projects, *i.e.*, OpenSSL (The OpenSSL Project, 1998), Bitcoin (Bitcoin Core developers, 2009), RocksDB (Facebook, 2012), and LLVM (The LLVM Project, 2003). They span diverse application domains, *i.e.*, security, blockchain, database management, and compiler infrastructure, thereby ensuring COMBENCH's broad representativeness of real-world development scenarios and the diverse error types.

To rigorously evaluate the effectiveness of current LLM-based ACER techniques against these real-world compilation errors, we conduct comprehensive experiments involving 12 popular LLMs of varying parameter scales. We explore two operational settings: *direct repair with perfect localization*, which assesses an LLM's core patch generation capabilities by abstracting away the complex task of fault localization; and *agent-based repair*, which evaluates the LLM's ability to autonomously reason, localize, and iteratively refine repairs. Beyond mere *compilation success*, our evaluation metrics are designed to provide a nuanced understanding of repair quality. We include *semantic correctness* to ensure that any generated fix preserves the original program's intended functionality, and *exact match* with ground-truth developer fixes to assess alignment with human-expert solutions. Our experiments reveal the varying capabilities of these LLMs for the ACER task, highlight crucial

limitations in current agent designs when faced with real-world complexities, and provide an in-depth analysis of their repair performance across different error types. Specifically, GPT-5 (OpenAI, 2025c) consistently demonstrates the highest overall repair capability, while Gemini-2.5-Pro (Google, 2025) exhibits superior semantic understanding capability. Agent-based approaches showed higher semantic correctness compared to direct repair; however, current designs still require further improvements.

In summary, this work makes the following major contributions:

- We introduce COMBENCH, the first repository-level, reproducible real-world benchmark for compilation error repair. COMBENCH addresses critical limitations of existing datasets by offering authentic errors sourced from GitHub CI with full project context, comprehensive error types, and rich annotations including ground-truth developer fixes.
- We develop a novel, robust framework for benchmark construction, which systematically overcomes challenges in collecting real-world compilation errors. This includes techniques for precise error extraction from noisy CI logs, causal inference of minimal repair patches from complex Git histories, and accurate reproduction of historical build environments.
- We conduct a comprehensive empirical evaluation of 12 popular LLMs based on COMBENCH. Our evaluation reveals varying LLM capabilities, highlights limitations in current agent designs, and provides novel insights into their repair performance across diverse error types.

2 BACKGROUND

Automated Compilation Error Repair (ACER). ACER is a sub-area of automated program repair (Huang et al., 2024) that targets compilation failures, *i.e.*, violations of a language's syntax or static type rules that prevent a successful build. Unlike logical bugs (Xia et al., 2025; Ruan et al., 2025; Zhang et al., 2024b; Yin et al., 2024; Xia & Zhang, 2024) or security vulnerabilities (Fu et al., 2022; Zhou et al., 2024; 2025; Zhang et al., 2024a), compilation errors are detected deterministically by the compiler and gate all downstream development, testing, and deployment, making their automated repair both tractable and high-value. Given a non-compiling program and its compiler diagnostics, the goal of ACER is to produce a patch that not only restores compilability but also preserves the developer's intended semantics. We discuss the inherent challenges of this task (*e.g.*, symptom–root-cause gaps, ambiguous repair strategies, and global consistency) in Appendix B.

Existing ACER Methods and Benchmarks. Recent ACER research is dominated by deep-learning-based approaches (Gupta et al., 2017; Mesbah et al., 2019; Li et al., 2023; Wu et al., 2024; Deligiannis et al., 2025) that learn from error–fix pairs, and the architectures of these methods are tightly coupled to the datasets on which they are trained and evaluated. However, mainstream benchmarks (Gupta et al., 2017; Ahmed et al., 2018) deviate from real-world software development in three ways: (1) *data granularity and context*—a decontextualized, single-file paradigm that fails to capture repository-level complexities; (2) *source authenticity and diversity*—instances exclusively drawn from novice student assignments, with DeepFix further lacking ground-truth patches; and (3) *task modeling bias*—a local sequence-to-sequence framing that overlooks global fault localization. As a consequence, models validated on these benchmarks have questionable generalization to real, complex projects, motivating a repository-level benchmark that addresses all three deficiencies. A complete account of method evolution and benchmark limitations is provided in Appendix C.

GitHub Actions as a Data Source. GitHub Actions (github, 2025a) is a Continuous Integration/Continuous Delivery (CI/CD) platform that automates build, test, and deployment pipelines directly within repositories, making it a valuable source for collecting real-world, repository-level compilation errors at scale. A detailed description of its workflow/job/step hierarchy and the runner-environment opacity that complicates failure reproduction is provided in Appendix E.

3 COMBENCH CONSTRUCTION

Goal. Existing ACER benchmarks often under-approximate real-world compilation failures by (i) discarding repository context, (ii) lacking trustworthy ground truth, and (iii) being hard to reproduce. COMBENCH is designed to close this gap by constructing *execution-verified repository-level* error–fix instances mined from real CI histories. Concretely, our construction must satisfy three requirements simultaneously: (1) *authenticity* (errors and fixes originate from developers' commits), (2) *ground*

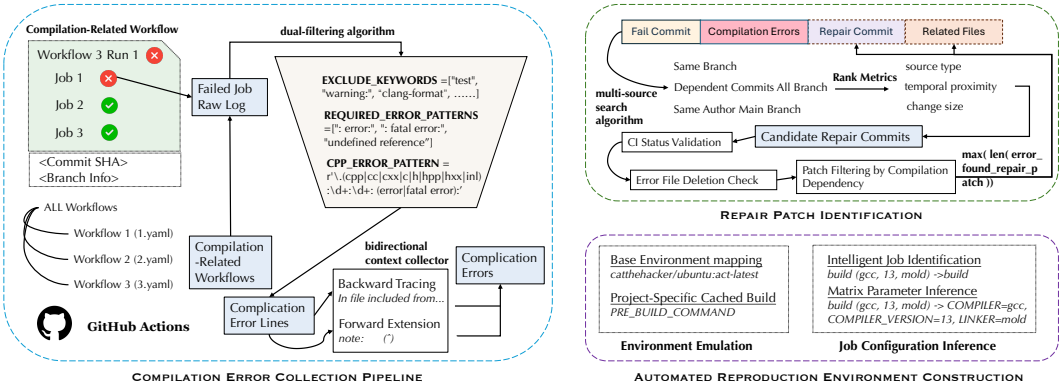


Figure 1: Overview of COMBENCH construction workflow: (1) **Error Collection** via dual-filtering and context tracing; (2) **Patch Identification** through multi-source search and dependency-based filtering; and (3) **Environment Construction** using Docker-based emulation and job configuration inference for automated reproduction with `act`.

truth (a causally aligned fixing change is identified and minimized), and (3) *reproducibility* (the original CI failure can be replayed locally under a faithful environment).

Overview. As illustrated in Figure 1, we build COMBENCH in three stages. These stages can be grouped into two primary phases: (i) **data collection** (Stages 1–2), which follows a three-step pipeline of CI failure collection, compilation error extraction, and repair patch identification (Sections 3.3–3.4); and (ii) **reproduction environment construction** (Stage 3), which builds a high-fidelity local environment and performs end-to-end *fail–fix–pass* execution verification (Section 3.5). The resulting benchmark consists of reproducible, repository-level compilation errors, each paired with a developer-provided ground-truth fix. It facilitates the evaluation of global repair capability, which we assess using compilation success and intended-behavior consistency (Section 5.4). We further analyze COMBENCH to characterize its representativeness and coverage (Section 4).

3.1 BENCHMARK INSTANCE DEFINITION AND DESIGN CONSTRAINTS

To make the construction objective explicit, we define a COMBENCH instance as a tuple:

$$\mathcal{I} = \langle \mathcal{R}, c_f, j_f, \mathcal{E}, c_r, \Delta_r, \mathcal{A} \rangle,$$

where $\mathcal{R}$ is a Git repository; c_f is the *failure commit* that triggers a CI job j_f to fail; $\mathcal{E}$ is the set of extracted compilation errors (each with compiler message, file/line, and contextual traces); c_r is the *repair commit* that causally fixes the failure; Δ_r is the *minimal compilation-relevant patch* derived from c_r; and $\mathcal{A}$ is a reproduction artifact (Docker image, workflow, and inferred job parameters) that enables replaying the failure and validating fixes.

This definition highlights two key constraints that guide the rest of this section. First, c_r must be *causally aligned* with c_f, rather than being a later revision that happens to pass CI. Second, both the failure and the fix must be *execution-verifiable* under $\mathcal{A}$: applying Δ_r to c_f should make the target job compile successfully.

3.2 DATA SOURCE SELECTION

The first step is selecting data sources that are likely to surface repository-level compilation failures. To ensure COMBENCH is realistic and captures repository-level properties, we selected large-scale, actively-maintained C/C++ projects from GitHub (github, 2025b). We chose C/C++ for its prevalence and complex build processes, and GitHub for its complete version histories and CI build logs. We further refine candidate projects using three criteria: (1) substantial codebase size (100K–1M LOC); (2) high development activity (dense commit history and frequent CI runs); and (3) mature and stable CI/CD (cic, 2025) pipelines. The selected projects are detailed in Section 4.

Table 1: The Eight Major Error Categories in Our Taxonomy, Ordered by Matching Priority (*Warning as Error* is excluded from the released benchmark).

Category	Description
Warning as Error	Warnings that are elevated to errors by compiler flags (*e.g.*, `-Werror`).
Declaration Error	Problems with undefined, redeclared, or conflicting identifiers (*e.g.*, symbols, namespaces).
Type Error	Violations of the C++ type system, such as mismatches, invalid conversions, or incomplete types.
Syntax Error	Issues related to code grammar and structure, like unexpected tokens or missing punctuation.
Template Error	Errors specific to C++ templates, such as instantiation or argument deduction issues.
Member Error	Issues with accessing or defining members of classes, including access specifier violations.
Function Error	Errors in function calls, overloads, constructors, or argument mismatches.
Semantic Error	Errors related to code logic and meaning, such as invalid operator usage or static assertions.

3.3 Compilation Error Collection Pipeline

Our goal in this stage is to recover $\mathcal{E}$, a structured set of compilation diagnostics, from raw CI logs. We address the challenges of noisy, heterogeneous logs with two modules. The first module *locates compilation-related failed jobs*: we parse workflow YAML files to keep only those invoking compilers or build tools (*e.g.*, `make`, `cmake`, `g++`, `ninja`), retrieve their failed runs within a recent time window, and download the raw logs of the failed jobs together with metadata (commit SHA, branch, workflow/job name). The second module *extracts structured error records* from these logs via a staged strategy: a *dual-filtering* step retains a line as a candidate only if it (i) does *not* match exclusion keywords (*e.g.*, `warning:`, `test`) and (ii) matches at least one compiler-error pattern (*e.g.*, `file:line:col: error:`); a *bidirectional context collector* then expands around each candidate—collecting backward `In file included from. . .` traces and forward `note: /` caret-marked lines—while skipping build-system noise and stopping at the next independent diagnostic. Each error is finally tagged with a rule-based type label and deduplicated at the commit level. The full details are provided in Appendix F.

We label each extracted error using a rule-based taxonomy. Our taxonomy contains *eight* categories (Table 1); because *Warning as Error* is typically project-policy-specific, we exclude it from the released benchmark, so the rest of the paper reports *seven* categories.

3.4 Repair Patch Identification

Given a failure commit c_f and its diagnostic set $\mathcal{E}$, the objective of this stage is to identify the corresponding developer repair commit c_r and to isolate a minimal patch Δ_r that is plausibly responsible for eliminating the observed compilation failures. We address this in three steps. First, we build a candidate set via a *multi-source search* that combines (i) subsequent commits on the originating branch, (ii) commits for which c_f is an ancestor (discovered via `git rev-list -ancestry-path`, capturing merges/rebases/cherry-picks), and (iii) same-author commits on `main`/`master`; candidates are ranked by source type, temporal proximity, and change size. Second, we *validate* candidates by requiring CI success and discarding fixes that simply delete the error-producing file. Third, for *dependency-based minimization*, let f be the primary error file and $D(f)$ its include closure; we keep a candidate only if its modified file set intersects $\{f\} \cup D(f)$, and define Δ_r as the diff restricted to that intersection. Among validated candidates we select c_r by maximizing the subset of $\mathcal{E}$ eliminated. The full details are provided in Appendix G.

3.5 Automated Reproduction Environment Construction

This stage constructs the reproduction artifact $\mathcal{A}$ to replay the original failure and validate fixes. The core challenge is that a CI workflow YAML is a *declarative* specification, while a failure occurs in a *concrete* runtime environment with implicit parameters (runner image, matrix expansion, environment variables). We address this with three components. *State restoration* clones $\mathcal{R}$ and checks out c_f

Table 2: Performance across different LLMs on COMBENCH. We report Compile Success (CS), Semantic Correctness (SC), and Exact Match (EM). **Bold** indicates the best performance in each category.

Model	bitcoin (12)			openssl (19)			rocksdb (64)			llvm (105)			Total (200)		
	CS	SC	EM	CS	SC	EM	CS	SC	EM	CS	SC	EM	**CS**	**SC**	**EM**
Proprietary Models															
GPT-5	**9**	**3**	1	**15**	8	6	**41**	**23**	12	**81**	**48**	21	**146**	**82**	40
GPT-4.1	4	2	0	12	7	6	37	20	12	61	35	15	114	64	33
Claude-Sonnet-4	7	**3**	1	12	8	6	36	22	**13**	59	35	21	114	68	41
Claude 3.7 Sonnet	7	1	0	10	5	5	35	19	11	55	32	16	107	57	32
Gemini-2.5-Pro	3	2	0	14	**11**	**11**	34	16	11	65	44	**26**	116	73	**48**
Open-source Models															
DeepSeek-V3	5	2	1	12	6	4	28	13	10	61	34	14	106	55	29
Qwen3-235B-A22b	5	1	1	13	8	6	23	10	7	58	30	12	99	49	26
Qwen3-30B-A3b	4	2	1	9	6	5	20	9	5	48	23	15	81	40	26
Qwen3-32B	2	2	1	12	4	4	19	9	6	47	27	11	80	42	22
Qwen3-14B	2	1	1	10	5	5	17	8	5	45	29	11	74	43	22
Qwen3-8B	1	1	0	9	4	6	18	8	2	52	29	11	80	42	19
Qwen3-4B	1	1	1	8	4	4	22	8	5	38	17	8	69	30	18
Agentic Framework															
Claude-Sonnet-4 †	6	3	1	13	8	8	40	21	14	72	52	29	131	84	52
Qwen3-32B †	1	1	0	9	4	5	21	10	7	43	23	8	74	38	20

† Evaluated within the OpenHands agentic framework.

to recover the exact failing source state. *Docker-based environment emulation* adopts a two-layer strategy: a base layer maps logical runner labels (*e.g.*, `ubuntu-latest`) to version-locked public images to bypass inaccessible proprietary runners, and a project-specific layer overlays curated `PRE_BUILD_COMMAND`s to install missing dependencies. *Job configuration inference* recovers the concrete job that produced the failure: it enumerates job instances via `act -l` together with matrix definitions, maps the human-readable job name in CI logs back to a YAML job ID using prefix matching with a Levenshtein-distance fallback, and infers matrix assignments by maximizing token coverage between the observed name and candidate values; the result is materialized as `-matrix` flags for `act` (nektos, 2025). The full details are provided in Appendix H.

3.5.1 END-TO-END EXECUTION VERIFICATION

Finally, we enforce an execution-level validity criterion for each candidate instance. We replay the *same* failing job j_f at c_f under $\mathcal{A}$ and require that it reproduces the extracted diagnostics. We then apply Δ_r (a clean cherry-pick restricted to the selected files) and re-run j_f under the same environment. An instance is kept only if the job reaches a successful compilation outcome (*fail–fix–pass*) and the originally extracted diagnostics no longer appear in the re-run logs. Only instances that satisfy this criterion are retained in the released COMBENCH. All retained instances are further cross-validated by three experts to ensure quality.

4 COMBENCH STATISTICS AND ANALYSIS

We analyze COMBENCH to characterize its representativeness and coverage. The detailed project characteristics, error type distribution, and phase-wise data entry are provided in Appendix M.

Key findings: COMBENCH contains 200 compilation errors from 4 large-scale C/C++ projects (Bitcoin, OpenSSL, RocksDB, LLVM), collectively spanning over 1.4 million lines of code and 69,500 files. Each compilation error has an average of 204 dependent files, confirming that these are complex, repository-level problems. The benchmark covers all 7 main error categories, with declaration errors (85) being the most prevalent, followed by type errors (41) and member errors (25).

5 EXPERIMENT

To comprehensively evaluate the ACER capabilities of LLMs, we designed an experimental framework. This framework employs three distinct metrics to assess the quality of generated patches, two experimental settings to probe different aspects of LLM performance (from pure patch generation to autonomous debugging), and 12 diverse models of varying sizes.

5.1 EVALUATION METRICS

We adopt three hierarchical metrics to evaluate generated patches, moving from basic syntax validity to deep semantic alignment.

Compile Success (CS): This metric measures whether the patch generated by the model successfully resolves the compilation error in the failing CI build without introducing new ones. However, some patches may achieve compilation success by inadvertently altering program semantics, which can lead to unexpected or incorrect program behaviors.

Semantic Correctness (SC): To address the risk of "shallow fixes" (*i.e.*, patches that compile but alter intended logic), SC evaluates whether the repair's semantics align with the developer's intent. It goes beyond mere compilation to assess the functional correctness of the fix. Notably, semantic inconsistency can arise for any type of compilation error, not only the *semantic* error category. This metric is measured by manual analysis (see Appendix D for the annotation protocol).

Exact Match (EM): For patches that successfully compile, this metric examines whether the generated patch is subsumed by the ground-truth repair after normalization (*i.e.*, removing whitespace and comments). This stringent measure captures the model's ability to reproduce developer-level fixes.

5.2 BASELINES

While traditional ACER tools are valuable, we prioritize SOTA LLM baselines as they are the most directly comparable under our repository-level setting. We evaluate under two complementary settings. **Setup 1: Repair with Perfect Fault Localization (PFL)** isolates the model's core patch-generation capability by providing it with the complete local context needed to fix the bug—specifically, the full source of the failing file and of the files modified by the ground-truth fix, the raw compiler diagnostics (error line, location, and notes), and an instruction to emit a search-and-replace patch in a single shot. **Setup 2: Agent-based Iterative Repair** instead evaluates end-to-end debugging in an interactive, multi-turn environment built on the OpenHands (Wang et al., 2025) framework: the agent starts from only the raw compiler error log and must autonomously explore the repository, execute shell commands, inspect files, and iteratively apply fixes until the build passes (capped at 100 iterations). Full prompt templates and tool configurations are provided in Appendix J.

5.3 MODELS

To comprehensively evaluate the capabilities of modern LLMs on the ACER task, we selected 12 popular models, spanning both open-source and closed-source families of varying sizes, to serve as our baselines. For the closed-source models, we utilized their official APIs (OpenAI, 2025a; Anthropic, 2025a; Google, 2025). For the open-source models, we leveraged the API provided by Deep Infra (Infra, 2024). For all models, we use `temperature=0.2` to produce high-quality answers (see Appendix J.3 for justification). The full list of models with their sizes, context lengths, and maximum output tokens is provided in Appendix I (Table 4).

5.4 RESULTS

5.4.1 OVERALL PERFORMANCE

The overall experiment results are shown in Table 2, across all 12 models, 4 repositories, and two settings. Analyzed on them, we derive the following findings:

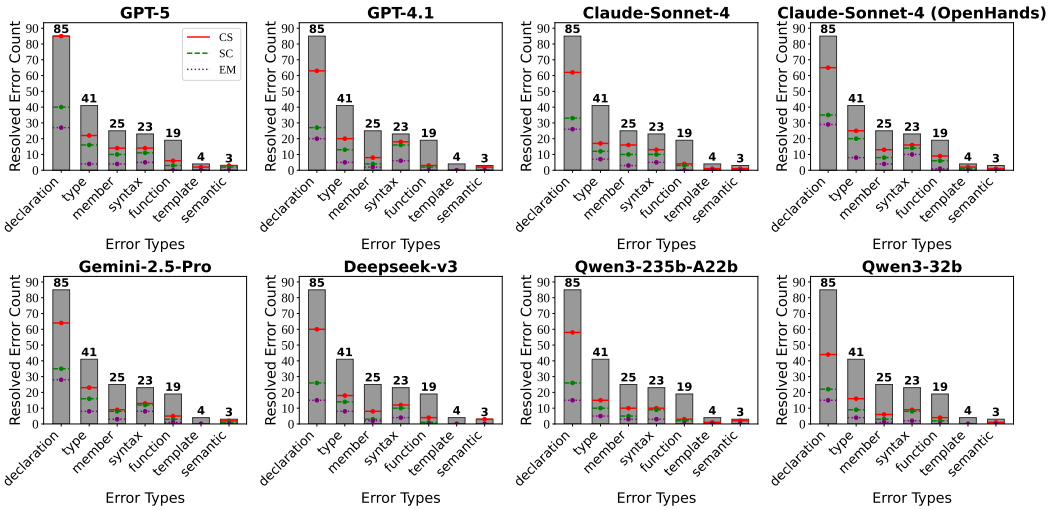


Figure 2: Performance of All LLMs Across Different Types

Finding 1: The "Compile-but-Incorrect" Gap is Systematic. Across all models, we observe a consistent drop-off from CS to SC, highlighting that satisfying compiler diagnostics is often easier than preserving repository-level semantics. This gap is most evident for top-tier models: GPT-5 achieves high CS rate (73%), yet only a subset of these compilable patches are semantically correct (SC 41%), and even fewer exactly reproduce the developer repair (EM 20%). We group these CS-but-not-SC failures into five recurring types (detailed in Appendix K): violations of semantic contracts, configuration-semantic boundary violations, protocol deviations, resource/policy mismatches, and convention bypass.

Finding 2: Performance Generally Scales with Model Size, but Plateaus in the Mid-range. There is a positive correlation between model size and repair capability, particularly within the Qwen3 family. The largest model, Qwen3-235B, achieves a CS of 99. However, performance appears to plateau among mid-sized models, with the 8B, 14B, 30B, and 32B variants exhibiting similar behavior across CS and SC. This suggests that, beyond a threshold, scaling alone is insufficient to close the CS–SC gap without better repository-level reasoning and convention awareness.

Finding 3: Models Exhibit Error-type Sensitivity. Beyond general performance rankings, we analyze each model's performance across our predefined error categories, presented in Figure 2. Different top-tier models show distinct strengths: Gemini-2.5-Pro is more effective on *type* errors, while Claude-Sonnet-4 demonstrates stronger performance on *member* errors. This indicates that ACER provides a fine-grained probe for different reasoning skills.

5.4.2 Agent vs. PFL

Overall, within the OpenHands agentic framework, Claude-Sonnet-4 achieves notably higher CS and SC than PFL (see Table 2), indicating that iterative exploration and context gathering can yield better end-to-end repair outcomes for strong models. Meanwhile, the agentic paradigm imposes a capability threshold: for Qwen3-32B, overall performance under the agentic framework decreases compared to PFL, suggesting that smaller models are more susceptible to noise and trial-and-error costs.

Finding 4: Agentic Repair Improves Overall Performance, but the Gains Are Capability-dependent. Our analysis suggests that PFL and the agentic paradigm represent two complementary repair mechanisms. With perfect fault localization, PFL favors fast, low-overhead local edits around a single reported error site; the agent, through iterative retrieval and multi-turn validation, can integrate scattered repository-level context and thus better handle global repairs that require cross-file dependency reasoning and multi-location consistency. For Claude-Sonnet-4, this extra context and validation yields higher CS/SC/EM (see Table 2). However, the gains are not uniform: some categories can still drop; moreover, agentic repair demands stronger planning, retrieval, and tool-use competence, and may degrade for weaker models (*e.g.*, Qwen3-32B). Overall, the agentic framework

can deliver better end-to-end repair performance for strong models, but its gains depend on the model's agentic capability and execution stability.

Finding 5: Some Hard Compilation Errors Remain Unsolved Even for Top-tier Models. In the PFL setting, the CS-failure intersection of GPT-5, Gemini-2.5-Pro, and Claude-Sonnet-4 still contains 34 compilation errors. These hard cases are rarely single-line fixes; instead, they often require cross-file and cross-configuration consistency. We summarize them into four recurring patterns: (1) API surface evolution causing declaration–definition–call-site inconsistencies; (2) access-control and encapsulation constraints restricting feasible fixes; (3) the complexity of template instantiation and overload resolution; and (4) coupled, cascading errors requiring coordinated fixes across translation units.

We conducted a detailed case study on instances where the best-performing model, GPT-5, in the PFL setting generated patches that compiled successfully but were semantically incorrect. We group these failures into two recurring patterns: violation of repository conventions and shallow substitution. Detailed case analyses are provided in Appendix K.

6 Related Work

Automated Dataset Construction and Mining Frameworks. Beyond ACER-specific benchmarks, software engineering has established various automated dataset construction and mining paradigms (Song et al., 2023; Sliwerski et al., 2005; Tang et al., 2025; Jimenez et al., 2024; Jiang et al., 2022; Song et al., 2022; Just et al., 2014). While these frameworks provide systematic ways to collect bug-fix pairs, their design choices differ from ComBench in ways that matter for compilation failures in modern CI/CD. First, regarding *data visibility*, most prior pipelines mine permanent artifacts such as commit histories or PR links (Song et al., 2023; Sliwerski et al., 2005; Jiang et al., 2022; Song et al., 2022; Jimenez et al., 2024). This introduces survivorship bias: it mainly captures bugs that survived the development process and were eventually merged, and consequently misses pre-merge compilation failures blocked by CI gatekeeping. ComBench addresses this by mining transient CI/CD logs to capture such failures that never enter the permanent repository history. Second, in terms of the *verification oracle*, existing benchmarks typically rely on test-driven oracles (*e.g.*, SWE-bench and Defects4J (Jimenez et al., 2024; Just et al., 2014)) and assume that a functional execution environment is already provided. In contrast, CI compilation failures are often environment-sensitive and require specific toolchains, dependencies, and build flags, which typically must be inferred and reconstructed from CI logs and configuration. ComBench adopts a *construction-as-validation* principle: it treats the executable fail–fix–pass compile cycle as the oracle and validates repairs in a reconstructed Docker environment.

Existing ACER Benchmarks. As discussed in Section 2, widely used ACER benchmarks such as DeepFix (Gupta et al., 2017) and TRACER (Ahmed et al., 2018) are constructed from decontextualized, single-file programs, typically from student assignments. This setting encourages local, snippet-level repairs with limited repository context, and it does not reflect CI failures that involve cross-file dependencies. In contrast, ComBench provides repository-level snapshots, real CI-triggered compilation failures, and developer-provided fixes, enabling evaluation in complex, real-world scenarios. A detailed attribute-level comparison against these datasets is given in Appendix L.

7 Conclusion

In this paper, we construct ComBench, a repository-level, real-world benchmark with comprehensive error types, reproducible environment, and a reusable data collection pipeline, which enables continuous evolving. Collected from 4 large-scale C/C++ projects across different domains, covering all 7 main categories of compilation errors, our benchmark is able to represent the real-world development scenario. Our benchmark evaluates both the compilable and sematic correct of the ACER tool's generated patches. These three metrics we designed reveal the error repair capabilities of each model from distinct perspectives and provide valuable insights and a solid evaluation foundation for future works in the ACER area.

Ethics Statement

We discuss the ethical aspects of this work from the following perspectives.

Data Collection and Privacy. All data in ComBench is collected from publicly available open-source repositories on GitHub. The compilation errors, developer commits, and CI logs we utilize are already publicly accessible. We do not collect any private or personally identifiable information. Developer identities are not exposed in the benchmark, as we only retain commit SHAs, file contents, and compiler diagnostics.

Intellectual Property and Licensing. The four projects in our benchmark (OpenSSL, Bitcoin, RocksDB, LLVM) are all released under permissive open-source licenses (Apache-2.0, MIT, BSD-3-Clause, and Apache-2.0 with LLVM exceptions, respectively). Our use of their public CI logs and version histories for research purposes is consistent with their licensing terms.

Dual-Use Concerns. While ComBench is designed to advance automated compilation error repair, the techniques evaluated on our benchmark could potentially be misused, *e.g.*, to automatically patch code in ways that introduce subtle vulnerabilities rather than fix them. We believe the benefits of improved ACER tools outweigh this risk, as the same capabilities also enable better detection of incorrect patches through our semantic correctness evaluation framework.

Environmental Impact. Our experiments involving 12 LLMs across 200 benchmark instances consumed non-trivial computational resources. We mitigated this by using `temperature=0.2` for deterministic outputs, avoiding unnecessary repeated runs, and reporting all results to maximize the utility of each evaluation cycle.

Reproducibility Statement

We are committed to open science by making ComBench publicly available, including all Docker environments, data collection pipelines, and evaluation scripts. Every released instance is execution-verified: Section 3.5 describes the reproduction-environment construction, and Section 5.4 defines the three evaluation metrics. The construction pipeline is detailed in Appendices F–J, covering compilation error collection, repair patch identification, environment emulation, the exact experimental setups for both the PFL and agent-based settings, and the annotation protocol used for semantic correctness (Appendix D). The full model list with decoding configurations (`temperature=0.2`) is given in Appendix I. Together these enable the community to reproduce our results, independently verify benchmark quality, and build upon our work.

References

Chardet: Python character encoding detector, 2025. https://github.com/chardet/chardet.

Ci/cd in wikipedia, 2025. https://en.wikipedia.org/wiki/CI/CD.

Continuous integration in wikipedia, 2025. https://en.wikipedia.org/wiki/Continuous_integration.

Compilation error in wikipedia, 2025. https://en.wikipedia.org/wiki/Compilation_error.

Ninja in wikipedia, 2025. https://en.wikipedia.org/wiki/Ninja.

Umair Z. Ahmed, Pawan Kumar, Amey Karkare, Purushottam Kar, and Sumit Gulwani. Compilation error repair: For the student programs, from the student programs. In *Proceedings of the 40th International Conference on Software Engineering: Software Engineering Education and Training*, pp. 78–87. ACM, 2018. doi: 10.1145/3183519.3183529.

DeepSeek AI. Deepseek-v3: The age of moe with 128k context window. https://www.deepseek.com/blog/deepseek-v3, 2025.

All-Hands-AI. Openhands git repository. https://github.com/All-Hands-AI/OpenHands, 2025.

Anthropic. Anthropic api reference. https://docs.anthropic.com/en/api/getting-started, 2025a.

Anthropic. Claude 3.7 sonnet. https://www.anthropic.com/news/claude-3-7-sonnet, 2025b.

Anthropic. Claude sonnet 4. https://www.anthropic.com/claude/sonnet, 2025c.

Bitcoin Core developers. Bitcoin core. https://github.com/bitcoin/bitcoin, 2009.

Pantazis Deligiannis, Akash Lal, Nikita Mehrotra, Rishi Poddar, and Aseem Rastogi. Rustassistant: Using llms to fix compilation errors in rust code. pp. 3097–3109, 04 2025. doi: 10.1109/ICSE55347.2025.00022.

Facebook. Rocksdb. https://github.com/facebook/rocksdb, 2012.

Michael Fu, Chakkrit Tantithamthavorn, Trung Le, Van Nguyen, and Phung Dinh. Vulrepair: A t5-based automated software vulnerability repair. In *To appear in the ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2022.

github. https://github.com/features/actions, 2025a.

github. https://github.com/, 2025b.

Google. Gemini 2.5 pro. https://deepmind.google/models/gemini/pro/, 2025.

Google. Gemini API Models. https://ai.google.dev/gemini-api/docs/models/gemini, 2025.

Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish Shevade. Deepfix: fixing common c language errors by deep learning. In *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence*, pp. 1345–1351. AAAI Press, 2017.

Kai Huang, Zhengzi Xu, Su Yang, Hongyu Sun, Xuejun Li, Zheng Yan, and Yuqing Zhang. Evolving paradigms in automated program repair: Taxonomy, challenges, and opportunities. *ACM Computing Surveys*, 57(2), 2024. doi: 10.1145/3696450.

Deep Infra. Deep infra api. https://deepinfra.com/, 2024.

Yanjie Jiang, Hui Liu, Xiaoqing Luo, Zhihao Zhu, Xiaye Chi, Nan Niu, Yuxia Zhang, Yamin Hu, Pan Bian, and Lu Zhang. Bugbuilder: An automated approach to building bug repository. *IEEE Transactions on Software Engineering*, 49(4):1443–1463, 2022.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. SWE-bench: Can language models resolve real-world github issues? In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=VTF8yNQM66.

René Just, Darioush Friedl, Gordon Fraser, Christian Hammer, Andreas Franz, Malte Isberner, and Jan Peleska. 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*, pp. 437–440, 2014.

Xueyang Li, Shangqing Liu, Ruitao Feng, Guozhu Meng, Xiaofei Xie, Kai Chen, and Yang Liu. Transrepair: Context-aware program repair for compilation errors. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. Association for Computing Machinery, 2023. doi: 10.1145/3551349.3560422.

Ali Mesbah, Andrew Rice, Emily Johnston, Nick Glorioso, and Edward Aftandilian. Deepdelta: learning to repair compilation errors. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 925–936. Association for Computing Machinery, 2019. doi: 10.1145/3338906.3340455.

nektos. https://github.com/nektos/act, 2025.

OpenAI. Openai api. https://openai.com/api/, 2025a.

OpenAI. Gpt-4.1. https://openai.com/index/gpt-4-1/, 2025b.

OpenAI. Gpt-5. https://openai.com/gpt-5/, 2025c.

Haifeng Ruan, Yuntong Zhang, and Abhik Roychoudhury. Specrover: Code intent extraction via llms. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, April 2025. doi: 10.1109/ICSE55347.2025.00080.

Hyunmin Seo, Caitlin Sadowski, Sebastian Elbaum, Edward Aftandilian, and Robert Bowdidge. Programmers' build errors: A case study (at google). In *Proceedings of the 36th International Conference on Software Engineering*, pp. 724–734. ACM, 2014. doi: 10.1145/2568225.2568255.

Jacek Sliwerski, Thomas Zimmermann, and Andreas Zeller. When do changes induce fixes? In *Proceedings of the International Workshop on Mining Software Repositories (MSR)*, 2005.

Xuezhi Song, Yun Lin, Siang Hwee Ng, Yijian Wu, Xin Peng, Jin Song Dong, and Hong Mei. Regminer: towards constructing a large regression dataset from code evolution history. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 314–326, 2022.

Xuezhi Song, Yijian Wu, Junming Cao, Bihuan Chen, Yun Lin, Zhengjie Lu, Dingji Wang, and Xin Peng. Bugminer: Automating precise bug dataset construction by code evolution history mining. In *Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering (ASE '23)*, pp. 1919–1929, 2023. doi: 10.1109/ASE56229.2023.00201.

Lingxiao Tang, Jiakun Liu, Zhongxin Liu, Xiaohu Yang, and Lingfeng Bao. LLM4SZZ: Enhancing SZZ Algorithm with Context-Enhanced Assessment on Large Language Models, April 2025.

The LLVM Project. The llvm compiler infrastructure. https://github.com/llvm/llvm-project, 2003.

The OpenSSL Project. Openssl. https://github.com/openssl/openssl, 1998.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, Hoang H. Tran, Fuqiang Li, Ren Ma, Mingzhang Zheng, Bill Qian, Yanjun Shao, Niklas Muennighoff, Yizhe Zhang, Binyuan Hui, Junyang Lin, Robert Brennan, Hao Peng, Heng Ji, and Graham Neubig. Openhands: An open platform for ai software developers as generalist agents, 2025. URL https://arxiv.org/abs/2407.16741.

Zhenyu Wu, Deheng Yang, Yan Lei, Huan Xie, Minghua Tang, and Maojin Li. Labelrepair: Sequence labelling for compilation errors repair. In *2024 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 860–871. IEEE Computer Society, 2024. doi: 10.1109/SANER60148.2024.00094.

Chunqiu Steven Xia and Lingming Zhang. Automated program repair via conversation: Fixing 162 out of 337 bugs for $0.42 each using chatgpt. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 819–831, 2024.

Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. Agentless: Demystifying llm-based software engineering agents, 2024. URL https://arxiv.org/abs/2407.01489.

Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. Demystifying llm-based software engineering agents. *Proceedings of the ACM on Software Engineering*, 2(FSE), 2025. doi: 10.1145/3715754.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, Chujie Zheng, Dayiheng Liu, Fan Zhou, Fei Huang, Feng Hu, Hao Ge, Haoran Wei, Huan Lin, Jialong Tang, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jing Zhou, Jingren Zhou, Junyang Lin, Kai Dang, Keqin Bao, Kexin Yang, Le Yu, Lianghao Deng, Mei Li, Mingfeng Xue, Mingze Li, Pei Zhang, Peng Wang, Qin Zhu, Rui Men, Ruize Gao, Shixuan Liu, Shuang Luo, Tianhao Li, Tianyi Tang, Wenbiao Yin, Xingzhang Ren, Xinyu Wang, Xinyu Zhang, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yinger Zhang, Yu Wan, Yuqiong Liu, Zekun Wang, Zeyu Cui, Zhenru Zhang, Zhipeng Zhou, and Zihan Qiu. Qwen3 technical report, 2025. URL https://arxiv.org/abs/2505.09388.

Xin Yin, Chao Ni, Shaohua Wang, Zhenhao Li, Limin Zeng, and Xiaohu Yang. Thinkrepair: Self-directed automated program repair. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 1274–1286, 2024.

Jian Zhang, Chong Wang, Anran Li, Wenhan Wang, Tianlin Li, and Yang Liu. Vuladvisor: Natural language suggestion generation for software vulnerability repair. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, pp. 1932–1944. ACM, 2024a. doi: 10.1145/3691620.3695555.

Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. Autocoderover: Autonomous program improvement. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 1592–1604. ACM, 2024b. doi: 10.1145/3650212.3680384.

Xin Zhou, Kisub Kim, Bowen Xu, Donggyun Han, and David Lo. Out of sight, out of mind: Better automatic vulnerability repair by broadening input ranges and sources. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. Association for Computing Machinery, 2024. doi: 10.1145/3597503.3639222.

Xin Zhou, Sicong Cao, Xiaobing Sun, and David Lo. Large language model for vulnerability detection and repair: Literature review and the road ahead. *ACM Transactions on Software Engineering and Methodology*, 34(5):1–31, 2025.

A Construction Challenges and Solutions

The construction of such a benchmark presents three significant challenges. First, *the heterogeneity and noise in CI logs* make it difficult to precisely extract structured compiler diagnostics. These logs lack a uniform format and are intermingled with massive irrelevant information, such as environment preparation, package installation, test results, and deployment details. Second, a more complex challenge lies in *the causal inference of the ground truth repair*, specifically locating the exact fix commit and isolating its minimal repair patch. This task is intricate because compilation failures often occur on ephemeral feature branches, complicating the direct traceability of fix commits and necessitating sophisticated search algorithms. Furthermore, identified fix commits frequently bundle the actual error repair with extensive, unrelated modifications like new features or refactoring. Precisely extracting the minimal, targeted patch from these multi-purpose changes is crucial for benchmark quality but inherently difficult. Finally, to ensure the benchmark's usability for evaluation, the critical challenge is *reproduction*: accurately recreating the exact historical CI environment in which the failure originally occurred. This is often complicated by two factors, *i.e.*, inaccessible proprietary runner images, which prevent proper container runtime environment setup, and ambiguous job configurations lacking essential execution parameters, thus hindering a precise reconstruction of the job's specific runtime environment.

Our ComBench framework systematically addresses these challenges through a targeted, multi-faceted approach. To identify compilation errors from noisy and heterogeneous CI logs, we employ a dual-filtering algorithm to prune irrelevant entries and validate error lines, complemented by a bidirectional context collector for detailed error information extraction. For causally inferring the ground truth repair, ComBench utilizes a multi-source search algorithm to locate and rank candidate repair commits across all branches. Following this, compilation dependency analysis precisely isolates the minimal repair patch by filtering for only the modified files directly implicated in resolving the error, thereby ensuring high-quality ground truth. Finally, we ensure the benchmark's reproducibility for evaluation through two mechanisms. For inaccessible proprietary runner images, a two-layer rebuild process maps the original build environment to a generic Docker image and then overlays project-specific pre-building commands. Concurrently, an intelligent matching algorithm addresses ambiguous job configurations by inferring job-specific parameters from human-readable names, enabling accurate reconstruction of the historical CI environment.

B ACER Challenges

The core task of ACER is to automatically generate a fix patch for a non-compiling program, utilizing the available compiler diagnostics (*e.g.*, error location and messages), such that the program not

only compiles successfully but also retains its intended semantics. However, despite the valuable diagnostic information provided by compilers, automatically generating a semantically correct and robust fix patch presents several significant challenges:

- *Symptom vs. Root Cause*: Compiler diagnostics often pinpoint symptoms rather than the underlying root cause. The actual problem might originate in a different part of the codebase than the reported error, necessitating a global understanding for accurate fault localization.
- *Ambiguity in Repair Strategies*: A single diagnostic message can correspond to multiple syntactically plausible repair strategies. Discerning between these and selecting the one that preserves intended semantics requires a deeper, global contextual understanding of the code's structure, data flow, and control flow.
- *Diverse Error Taxonomy*: Compilation errors encompass a wide spectrum of issues (*e.g.*, syntax errors, type mismatches, missing declarations). Each category often demands a distinct and tailored repair strategy, challenging the wide applicability of ACER tools.
- *Global Consistency*: A seemingly correct local fix might introduce new compilation errors or semantic deviations elsewhere. The challenge of maintaining global consistency and preventing cascading failures makes generating a robust and globally sound patch highly complex.

C ACER Methods and Existing Benchmark Limitations

In recent years, deep learning-based methods have become the main approach for ACER. These techniques leverage the powerful pattern recognition and sequence modeling capabilities of neural networks to automatically generate fix patches by learning from vast datasets of error-fix pairs. Their evolution reveals a clear path of continuously enriching input information, optimizing model architectures, and innovating problem modeling. Early approaches, such as DeepFix (Gupta et al., 2017), established a sequence-to-sequence foundation. Subsequent methods, including DeepDelta (Mesbah et al., 2019) and TransRepair (Li et al., 2023), integrated compiler feedback and code context. More recently, LabelRepair (Wu et al., 2024) optimized the process through sequence labeling. Recent work, such as RUSTASSISTANT (Deligiannis et al., 2025), has introduced an iterative repair loop that repeatedly queries an LLM and uses the resulting compiler feedback to progressively refine patches. All these learning-based methods evolve to progressive improvements in repair accuracy.

However, the architectures and capabilities of these learning-based methods are heavily influenced and constrained by existing datasets and benchmarks (Gupta et al., 2017; Ahmed et al., 2018). Since these methods are primarily trained and evaluated on single-file and context-limited datasets, they inherently only prioritize local fixes. As a result, their ability to handle cross-file dependencies and project-level compilation issues remains largely unexplored. This highlights how the evaluation dataset significantly shapes method design and underscores the necessity for new, more representative benchmarks.

Current mainstream benchmarks for ACER exhibit notable limitations in capturing the complexity of real-world development scenarios, which is revealed by significant discrepancies from real-world software development practices across three key aspects:

- *Data Granularity and Context*: Existing benchmarks, such as DeepFix and TRACER, adopt a decontextualized, single-file paradigm, providing only short code snippets (*e.g.*, avg. under 20 lines). This approach isolates error instances from their project context, thereby failing to capture repository-level complexities essential for real-world repair, such as cross-file symbol resolution, inter-module linking issues, and complex build system configurations.
- *Data Source Realism and Diversity*: Current benchmarks suffer from limited data sources, which are exclusively constructed from novice student programming assignments (*e.g.*, DeepFix (Gupta et al., 2017), TRACER (Ahmed et al., 2018)). This narrow origin leads to a homogeneous set of relatively simple error types (*e.g.*, missing delimiters or declarations). Thus, they fail to represent the diverse and complex errors prevalent in industrial software, which often stem from intricate system interactions in large-scale projects. Furthermore, DeepFix's evaluation benchmark notably lacks corresponding human-written ground truth patches, precluding the assessment of whether a proposed fix aligns with the developer's intended corrections.

- *Task Modeling Bias*: The aforementioned simplified data settings directly lead to a biased task modeling, where ACER is commonly framed as a local sequence-to-sequence problem (Gupta et al., 2017; Ahmed et al., 2018; Deligiannis et al., 2025). Models like DeepFix take the entire program as input (Gupta et al., 2017). TRACER, in contrast, focuses on a single erroneous line (Ahmed et al., 2018), and RUSTASSISTANT accepts isolated code snippets surrounding the locations in compiler diagnostics (Deligiannis et al., 2025). Despite these varied input approaches, these models implicitly assume that all necessary information to solve the problem is locally available. This contrasts sharply with real-world scenarios, where the core challenge of repair involves global fault localization within a large repository and the integration of scattered contextual information. Even the authors of TRACER acknowledge that their model cannot effectively handle errors requiring global context (Ahmed et al., 2018).

In summary, the interconnected limitations across data granularity, source authenticity, and task modeling create a significant gap between current ACER research and practical application. Consequently, repair models validated on existing benchmarks often have questionable generalization capabilities when applied to real, complex projects. Therefore, developing a repository-level benchmark, which addresses all these deficiencies, is crucial for advancing the field.

D Semantic Correctness Annotation Protocol

To rigorously evaluate whether a generated patch preserves the intended semantics of the original program, we design a structured annotation protocol for Semantic Correctness (SC). This section details the annotation criteria, process, annotator composition, disagreement resolution, and inter-annotator agreement.

D.1 Annotation Criteria

A generated patch is considered **semantically correct** if and only if it satisfies *all* of the following conditions:

1. **Compilation Success**: The patched program compiles without errors under the same CI environment as the original failure.
2. **Intent Preservation**: The fix preserves the developer's intended behavior. Specifically, the patched code should not alter the externally observable behavior (e.g., return values, side effects, error-handling paths, protocol-level outputs) in ways that deviate from the developer's ground-truth fix.
3. **No Unintended Side Effects**: The fix does not introduce new functional regressions or break existing functionality beyond the scope of the repair.
4. **Convention Compliance**: The fix respects the repository's coding conventions and architectural constraints (e.g., using project-specific headers instead of standard library alternatives when the project explicitly defines custom abstractions).

Negative Examples. We categorize semantically incorrect patches that nevertheless compile successfully into five recurring failure types:

- *Semantic Contract Violation*: Breaking API contracts, return-value semantics, or nullability guarantees (e.g., returning a generic error code instead of a domain-specific one).
- *Configuration-Semantic Boundary Violation*: Altering conditional compilation or feature semantics in a way that compiles but violates the project's intended build configuration logic.
- *Protocol Deviation*: Changing error-handling or communication paths that alter externally observable behavior (e.g., sending a misleading TLS alert instead of the correct one).
- *Resource/Policy Mismatch*: Introducing resource management or policy deviations that change runtime strategies.
- *Convention Bypass*: Implementing an alternative fix that bypasses repository abstractions and layering (e.g., using standard library `<span>` instead of project-specific `<span.h>`).

D.2 Annotation Process

The annotation proceeds in two phases:

Phase 1 — Independent Annotation. Three annotators independently evaluate each compiled patch. All annotators are authors of this paper with expertise in C/C++ software engineering and compilation systems. Each annotator is provided with:

- The original failure commit c_f and its compiler diagnostics.
- The developer's ground-truth patch Δ_r.
- The model-generated patch.
- The diff between the model-generated patch and the ground-truth patch.
- The full source code of all modified files (both before and after patching).

Each annotator assigns one of three labels:

- `SC=1` (Semantically Correct): The patch satisfies all four criteria above.
- `SC=0` (Semantically Incorrect): The patch violates at least one criterion.
- `Uncertain`: The annotator cannot confidently determine semantic correctness (e.g., due to insufficient domain knowledge about the specific project).

Phase 2 — Disagreement Resolution. For instances where not all three annotators agree, a majority vote is used as the final label. If the vote is tied (e.g., 1-1-1 or 1-1-0 with one `Uncertain`), the three annotators discuss the case together and reach a consensus. During discussion, annotators reference the ground-truth patch as the authoritative standard and evaluate whether the model-generated patch achieves the same semantic outcome, even if through a different implementation path. If consensus cannot be reached after discussion, the instance is labeled as `SC=0` (conservative), as the uncertainty itself indicates the patch does not clearly preserve intended semantics.

D.3 Inter-Annotator Agreement

We measure inter-annotator agreement using Fleiss' κ across all three annotators on the full set of compilable patches. The overall Fleiss' κ is **0.83**, indicating near-perfect agreement. Table 3 reports the per-model agreement statistics.

Table 3: Inter-Annotator Agreement for Semantic Correctness. We report the percentage of instances with full agreement (3/3), majority agreement (2/3), and Fleiss' κ per model.

Model	Full Agreement	Majority	Fleiss' κ
GPT-5	91.1%	97.3%	0.85
GPT-4.1	89.5%	96.5%	0.83
Claude-Sonnet-4	88.6%	96.5%	0.82
Claude 3.7 Sonnet	87.9%	96.3%	0.81
Gemini-2.5-Pro	90.5%	97.4%	0.84
DeepSeek-V3	88.7%	95.3%	0.82
Qwen3-235B-A22b	87.9%	95.0%	0.81
Qwen3-30B-A3b	86.4%	93.8%	0.79
Qwen3-32B	85.0%	93.8%	0.78
Overall	**88.4%**	**95.8%**	**0.83**

The high agreement scores indicate that the SC metric, while based on manual analysis, is applied consistently across annotators. The primary sources of disagreement are patches in unfamiliar project-specific domains (e.g., LLVM's pass infrastructure), where annotators initially differed on whether a fix respects repository conventions but reached consensus after discussion.

E GitHub Actions CI System Details

GitHub Actions (github, 2025a) is a Continuous Integration/Continuous Delivery (CI/CD) platform that automates build, test, and deployment pipelines directly within repositories. It serves as a valuable source for collecting real-world project-level compilation errors. Its automation is defined using YAML files stored in the repository's `.github/workflows/` directory, which specify the events that trigger workflows, the jobs to be executed, and the individual steps within those jobs.

The system operates with a clear conceptual hierarchy. A *Workflow* is the top-level unit, corresponding to an automated process, defined in a YAML file and initiated by repository events (*e.g.*, "push", "pull_request"). Each workflow consists of one or more *Jobs*. A job is a sequence of tasks that runs on a specific virtual machine (runner). Jobs can run in parallel or sequentially, and the build matrix (`strategy: matrix`) allows a single job definition to be expanded into multiple jobs under different configurations. Critically, the execution log generated by each job is a valuable source for collecting compilation errors. A *Step* is the smallest unit of work within a job, executing sequentially. Steps can run shell commands or execute reusable *Actions*, which encapsulate common tasks like checking out code (`actions/checkout`) or setting up toolchains (`actions/setup-python`).

However, this operational paradigm introduces a critical challenge: the *opacity* of runner environments. Although logs capture the compilation output, the internal state of the virtual machine, including pre-installed system libraries, environment variables, and transient filesystem configurations, remains opaque to researchers. This environmental invisibility significantly complicates the reproduction of environment-dependent compilation failures, as the source code repository alone often fails to encapsulate the comprehensive context required for a successful build.

F Compilation Error Collection Pipeline Details

F.1 CI Failure Data Collection

This module collects raw failure logs from GitHub Actions (github, 2025a), which serve as the primary evidence for compiler diagnostics. We adopt a top-down refinement procedure to ensure that the failures are compilation-related and that the retrieved logs are attributable to a specific job:

Identifying Compilation-Related Workflows. We parse CI workflow YAML files to identify workflows that contain explicit compilation commands or build tool invocations (*e.g.*, `make`, `cmake`, `g++`, `clang++`, `ninja` (nin, 2025)).

Locating Failed Workflow Runs. Within a predefined time window (*e.g.*, the last 90 days), we retrieve runs of these workflows that are marked as `failed`.

Pinpointing the Failed Job. For each failed workflow run, we analyze its constituent jobs. By examining the execution status of each job, we pinpoint the specific job responsible for the failure.

Extracting Raw Logs. For each failed job, we download the complete raw log, and store it together with metadata (repository, workflow/job name, triggering commit SHA, branch, timestamp). This structured storage decouples downstream analysis from the CI provider API and supports reproducible re-processing.

F.2 Compilation Error Extraction

This module transforms raw job logs into structured compilation error records. The design goal is to be robust to log noise while maintaining high precision on true compiler diagnostics. Accordingly, we adopt a staged extraction strategy that first localizes candidate error lines, then reconstructs multi-line diagnostic context, and finally assigns an error taxonomy label.

Log Preprocessing and Normalization. To handle potential encoding inconsistencies across different platforms, the pipeline first uses the `chardet` library (cha, 2025) to detect each log file's encoding and normalizes it to UTF-8. Concurrently, common noisy content, such as CI system timestamps, is removed using regular expressions to streamline subsequent pattern matching.

Identifying Candidate Error Lines (Dual Filtering). A line is considered a candidate if it (i) does *not* match any item in `EXCLUDE_KEYWORDS` (*e.g.*, `test`, `warning:`, `clang-format`), and (ii) matches

at least one item in `REQUIRED_ERROR_PATTERNS` (*e.g.*, `: error:`, `: fatal error:`, `undefined reference`). Candidates are further validated against `CPP_ERROR_PATTERNS`, which capture typical compiler formats (*e.g.*, `file.cpp:line:column: error:`).

Context Reconstruction. Since compiler diagnostics are often multi-line, we expand around each candidate to collect both (a) backward include traces (up to 20 lines, *e.g.*, `In file included from...`) and (b) forward diagnostic continuation (up to 15 lines, including `note:` lines and caret-marked snippets (^)). This process skips build-system noise (*e.g.*, `[ 3%] Building CXX object ...`) and terminates upon encountering the next distinct error or warning, ensuring the completeness and independence of each error's context.

Fine-Grained Error Taxonomy. To support subsequent analysis, we label each extracted error using a rule-based taxonomy. Our classification method matches each error message against a curated set of regular expressions distilled from a large corpus of real-world C/C++ compilation errors. To resolve ambiguities where one message matches multiple rules, we apply rules under a prioritized matching strategy, so that more specific and critical error types are identified first; the priority order is also given in Table 1.

Data Aggregation and Storage. To avoid double-counting repeated diagnostics triggered by multiple CI jobs for the same commit, we deduplicate errors at the commit level by merging jobs whose normalized diagnostic sets are identical. The diagnostic set is compared by keying on the extracted primary error messages and key fields such as file/line locations. All extracted information is stored in JSONL records with fields including `repo`, `commit_sha`, `branch`, `timestamp`, `workflow_name`, `job_names`, `error_lines`, `error_details`, and `error_types`.

G Repair Patch Identification Details

Given a failure commit c_f and its diagnostic set $\mathcal{E}$, the objective of this stage is to identify the corresponding developer repair commit c_r and to isolate a minimal patch Δ_r that is plausibly responsible for eliminating the observed compilation failures. This is non-trivial because CI failures often occur on ephemeral feature branches, where the immediate subsequent commits may be unavailable after merges and branch deletion, and fixing commits frequently entangle unrelated changes (features or refactors) that would pollute ground truth. We address this with a multi-source candidate search, followed by validation and dependency-based minimization.

G.1 Candidate Search and Ranking

Starting from c_f, we build a candidate set of potentially fixing commits using a *multi-source search* strategy:

- **Subsequent commits on the same branch:** commits after c_f on the originating branch (when the branch remains accessible), which typically corresponds to the most common repair path during local development.
- **Dependent commits across branches:** commits for which c_f is an ancestor, discovered via `git rev-list -ancestry-path`, covering merges, rebases, and cherry-picks. Since the candidate codebase directly evolves from the failure commit, this ancestor relation provides a causal relevance guarantee.
- **Same-author commits on the main branch:** commits by the same author on `main`/`master` after c_f, covering PR-merged fixes.

After deduplication, the collected candidates are ranked using a multi-criteria heuristic algorithm. The ranking prioritizes candidates based on: (1) source type (*same_branch* > *dependency* > *main_branch*), (2) temporal proximity to the failure (closer is better), and (3) change size (fewer changed files and lines are better). This strategy ensures that the most probable and relevant fixing commits are analyzed first.

G.2 Validation and Filtering

We apply three filters to isolate genuine fixes and extract a minimal patch.

CI Status Validation. We retain only candidates whose CI checks conclude success (*i.e.*, the corresponding workflow runs pass), to avoid choosing commits that merely shift the failure elsewhere.

Error File Deletion Check. We discard candidates that fix the failure by deleting the error-producing file, since such changes do not represent meaningful repair.

Dependency-Based Patch Minimization. For each remaining candidate, we compute the file-level diff and test whether the modified files intersect with the compilation dependency closure of the error-producing file. Let f be the primary error file extracted from $\mathcal{E}$. We build a dependency closure $D(f)$ by recursively parsing local `#include` directives to collect directly/indirectly included headers reachable from f. A candidate is considered a valid repair only if its modified file set $M(c)$ satisfies $M(c) \cap (\{f\} \cup D(f)) \neq \emptyset$. We then define Δ_r as the restriction of the commit diff to files in this intersection, ordered by proximity (f first, then direct includes, then transitive includes).

G.3 Best Patch Selection

We traverse validated candidates in ranking order and select c_r by a maximization principle: choose the commit that resolves the largest subset of the original errors in $\mathcal{E}$. We use early termination: once a candidate eliminates all extracted errors, the search stops.

G.4 Data Aggregation and Storage

We store validated error–fix pairs as JSONL records with `failure_commit`, `repair_commit`, `error_lines`, `error_details`, `workflow_name`, `job_names`, and `compilation_related_files`. Within each record, diagnostics are deduplicated by error message to ensure that only unique errors are retained.

H Reproduction Environment Construction Details

This stage constructs the reproduction artifact $\mathcal{A}$ to replay the original failure and validate fixes. The core challenge is that a CI workflow YAML is a *declarative* specification, while a failure occurs in a *concrete* runtime environment with implicit parameters (runner image, matrix expansion, environment variables). We therefore reconstruct both the code state and the runtime state, and then infer the precise job configuration that produced the failure.

H.1 State Restoration

We restore the repository to the exact historical state by cloning and checking out c_f (`git checkout`).

H.2 Environment Emulation

We emulate the CI runtime with a two-layer Docker strategy. The *base layer* maps logical runner labels (*e.g.*, `ubuntu-latest`) to version-locked public images (*e.g.*, `catthehacker/ubuntu:act-latest`), mitigating the inaccessibility of proprietary runner images. On top of this, the *project-specific layer* executes a curated `PRE_BUILD_COMMAND` to install missing dependencies (*e.g.*, `apt-get install -y build-essential cmake`) and caches the result via `docker commit`. This turns a one-time manual reconstruction effort into a deterministic setup process for subsequent reproductions.

H.3 Job Configuration Inference

With code and environment in place, we infer the concrete job configuration and execute it via `act` (nektos, 2025). The main difficulty is mapping the human-readable job name observed in CI logs (*e.g.*, `build (gcc, 13, mold)`) back to (i) the static job ID in YAML and (ii) the specific matrix assignment that instantiated that job. We perform three steps.

Intelligent Job ID Identification. We enumerate possible job instances by combining `act -l` with the workflow's matrix definitions. We then match the observed job name using a two-stage heuristic: exact prefix matching first, and a Levenshtein-distance-based similarity fallback when prefixes are ambiguous; we select the best-scoring instance under this similarity.

Table 4: Baseline LLMs Used in Our Evaluation

Open-source Models				Closed-source Models			
Model	**Size**	**Context**	**Output**	**Model**	**Size**	**Context**	**Output**
DeepSeek-V3 (AI, 2025)	671B	128K	8,192	GPT-5 (OpenAI, 2025c)	-	272K	128,000
Qwen3-235B-A22b (Yang et al., 2025)	235B (22B active)	128K	8,192	GPT-4.1 (OpenAI, 2025b)	-	1M	32,768
Qwen3-30B-A3b (Yang et al., 2025)	30B (3B active)	128K	8,192	Claude 4 Sonnet (Anthropic, 2025c)	-	200K	16,384
Qwen3 (4/8/14/32B) (Yang et al., 2025)	4/8/14/32B	128K	8,192	Claude 3.7 Sonnet (Anthropic, 2025b)	-	128K	16,384
				Gemini 2.5 Pro (Google, 2025)	-	1M	64,000

Matrix Parameter Inference. For `strategy: matrix` jobs, the execution is determined by a combination of variables. We parse the matrix keys and candidate values from YAML, tokenize the observed job name, and infer the concrete matrix assignment by maximizing token coverage. Specifically, for each matrix key, we choose the value whose string form appears in (or best matches) the tokenized job name; meanwhile, we validate the inferred key–value pairs against the enumerated instances from the previous step to avoid impossible combinations. The final assignments are materialized as `-matrix` flags for `act`.

Dynamic Command Synthesis and Execution. We synthesize a complete `act` command by assembling the workflow path (`-W`), the inferred job ID (`-j`), the trigger event (*e.g.*, `push`), the inferred `-matrix` parameters, runner-to-image bindings (`-P`), and required environment variables (`-env`, *e.g.*, `GITHUB_TOKEN`). We redirect both stdout and stderr to log files for debugging and auditing.

I Baseline Models

To comprehensively evaluate the capabilities of modern LLMs on the ACER task, we selected 12 popular models to serve as our baselines. Table 4 presents their names, sizes, context lengths, and maximum output tokens. For the closed-source models, we utilized their official APIs (OpenAI, 2025a; Anthropic, 2025a; Google, 2025). For the open-source models, we leveraged the API provided by Deep Infra (Infra, 2024). For all models, we use `temperature=0.2` to produce high-quality answers (see Appendix J.3 for justification).

J Experimental Setup Details

J.1 Setup 1: Repair with Perfect Fault Localization (PFL)

This experimental setup is designed to isolate and evaluate the LLM's core patch generation capability. The input prompt is constructed to provide the model with complete local context, including: (1) the full source code of the failing file and modified files within ground-truth fix; (2) the full, raw compiler diagnostic messages containing error line with error message and error-producing location with file, line, and column numbers, alongside with error detail containing note and code reference (^); and (3) a clear instruction to generate a patch. In this single-shot generation approach, the LLM is expected to output a single repair with cross-files multi-hunk fixes directly in a search-and-replace format, without multi-turn interaction. The evaluation, therefore, focuses purely on the model's effectiveness at generating a correct patch under these ideal input conditions. We retry one time if the LLMs provide no applicable search-replace patch.

J.2 Setup 2: Agent-based Iterative Repair

In contrast, this setup evaluates an LLM's capability to autonomously diagnose, localize, and repair compilation errors in an interactive, multi-turn environment. Specifically, we leverage the OpenHands (Wang et al., 2025) framework to implement this agent-based setup. The LLM acts as an agent within this framework, beginning with a minimal prompt that provides only the raw compiler error log. From there, it must reason about the problem and explore the entire codebase to solve it. The agent interacts with the environment using a predefined set of tools that allow it to inspect files (`fetch`), execute shell commands for compilation or searching (`execute_bash`), apply code modifications (`str_replace_editor`), manage its internal strategy (`think`, `task_tracker`), and signal completion (`finish`). The agent iteratively uses these tools, observes the outcomes, and

refines its approach until the error is resolved. This setup measures the LLM's comprehensive, end-to-end debugging skills. We retry one time if the agent provides no legal diff. Our configurations utilize version 0.55.0 of the OpenHands GitHub repository (All-Hands-AI, 2025), which was the latest version available at the time of writing this paper. Furthermore, the maximum number of iterations is set to 100, aligning with the default configuration of SWE-bench (Jimenez et al., 2024) in OpenHands.

J.3 Temperature Selection Justification

We use `temperature=0.2` across all experiments. This choice is motivated by three considerations: (1) Low temperatures (0.0–0.2) are widely adopted in code repair benchmarks (Jimenez et al., 2024; Wang et al., 2025; Xia et al., 2024) to ensure deterministic and reproducible outputs; (2) ACER targets compilable fixes rather than creative generation, making low temperature a task-appropriate choice; and (3) `0.2` preserves slight stochasticity for bounded exploration while avoiding the degeneration risks of higher values.

K Case Study Details

K.1 Failure Mode 1: Violation of Repository Conventions

Case Study 1 (Bitcoin): Custom vs. Standard Library Mismatch. A common LLM failure mode is proposing a locally plausible fix that violates repository conventions. For an error, `'std::span' has not been declared`, the LLM's patch added the standard header `#include <span>`. In contrast, the developer's ground-truth patch replaced a forward declaration (`- class Span;`) with a project-specific header (`+ #include <span.h>`). The LLM's patch, while syntactically plausible, is semantically incorrect. It introduces the standard C++20 `<span>` into a project designed around a custom, C++17-compatible `<span.h>`, violating its architectural constraints. This choice is incompatible with the project's custom data structures (*e.g.*, `prevector`) and can trigger compilation failures in other translation units or build configurations beyond the failing CI target. This case highlights that without a global understanding of a project's internal APIs, an LLM may default to a common but contextually invalid solution.

K.2 Failure Mode 2: Shallow Substitution

Case Study 2 (OpenSSL): Global Semantic Refactoring vs. Shallow Identifier Substitution. This case illustrates a common but flawed LLM repair strategy: resolving an undeclared identifier error by substituting the unknown identifier with an arbitrary, but defined, one from the nearby code, without regard for semantic correctness. Faced with the undeclared error constant `PROV_R_FAILED_DURING_DERIVATION`, even with the clear hint in comments `/* the public key was probably a weak key */`, the LLM's patch still replaced it with a nearby, generic constant, `ERR_R_INTERNAL_ERROR`, simply to satisfy the compiler. The developer's ground-truth patch, however, performed a deeper semantic correction, refactoring the entire error-handling block to use a more protocol-appropriate alert (`+ SSLfatal(..., SSL_R_BAD_KEY_SHARE)`). The LLM's substitution, while compilable, is semantically incorrect as it alters the program's externally observable behavior by sending a different and misleading TLS alert. This highlights the model's tendency to prioritize syntactic validity—finding any defined identifier of the correct type—over understanding the code's domain-specific logic.

Figure 3 shows the developer's ground-truth patch for this case.

L Existing ACER Datasets Comparison

Table 5 provides a detailed comparison of existing ACER benchmarks.

```
--- a/ssl/s3_lib.c
+++ b/ssl/s3_lib.c
@@ -5039,12 +5039,7 @@
       /*
        * the public key was probably a weak key
        */
-      if (ERR_GET_REASON(ERR_peek_last_error())
-          == ERR_LIB_EC) {
-          SSLfatal(s, SSL_AD_INSUFFICIENT_SECURITY,
-                   PROV_R_FAILED_DURING_DERIVATION);
-      } else {
-          SSLfatal(s, SSL_AD_INTERNAL_ERROR,
-                   ERR_R_INTERNAL_ERROR);
-      }
+      SSLfatal(s, SSL_AD_ILLEGAL_PARAMETER,
+               SSL_R_BAD_KEY_SHARE);
       goto err;
   }
```

Figure 3: Developer's ground-truth patch for the OpenSSL case study (Failure Mode 2).

Table 5: Comparison of Existing Datasets in ACER.

Attribute	**DeepFix Dataset** (Gupta et al., 2017)	**TRACER Dataset** (Ahmed et al., 2018)
Data Source	Introductory C programming course assignments.	Novice programmer assignments in CS1 courses.
Size	6,971 error instances.	16,985 error instances.
Format	Standalone single C files (avg. 17.87 lines).	Standalone Single C files (avg. 18.49 lines).
Error Types	Primarily syntax errors (e.g., missing delimiters) and declaration errors.	Primarily identifier errors, incorrect arguments, and missing statements.
Ground-Truth Fixes	**No.**	**Yes.**

M Data Statistics and Distribution Details

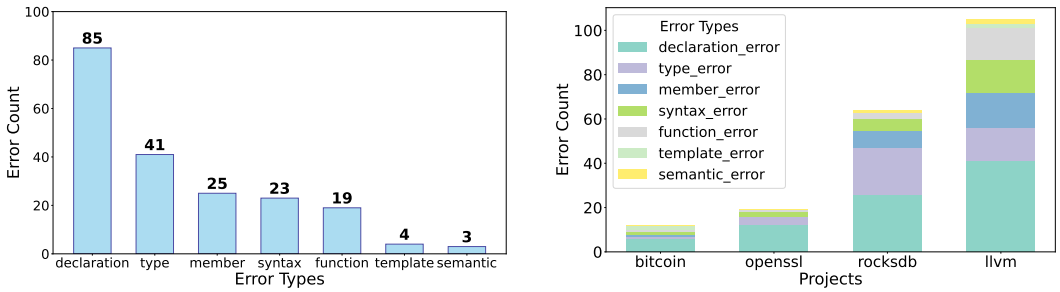
This section presents a detailed data analysis of ComBench, with the objective of demonstrating the representativeness of the benchmark of real-world, repository-level compilation errors. We first characterize the projects included in the benchmark (Table 6), then provide an in-depth examination of the distribution of compilation error types (Figure 4). We also present the data collected in each phase to illustrate the progressive filtering and reduction of data (Table 7).

M.1 Project Characteristics

Our ComBench benchmark collects errors from four large-scale C/C++ projects. Table 6 provides a detailed overview of these projects, including their main language, lines of code (LoC), number of code files, commits, workflow runs, and the total compilation errors identified. Specifically, for the LoC and file count metrics, only source files with common C/C++ extensions (`.c`, `.cpp`, `.cc`, `.h`, and `.hpp`) were considered. As shown in Table 6, the four projects collectively span over 1.4 million lines of code and 69,500 files. This substantial scale and complexity are representative of real-world software development. Among these, LLVM exhibits the highest number of commits and runs, consequently contributing the most errors (105) to the benchmark. A key observation from Table 6 is that each compilation error in our dataset has an average of 204 dependent files. This widespread dependency demonstrates that our benchmark's errors are not isolated, single-file issues but complex, repository-level problems. Consequently, ComBench is well-suited to evaluate an ACER tool's ability to perform the global analysis required for real-world error resolution. Furthermore, the benchmark includes a mix of C (OpenSSL) and C++ projects (Bitcoin, RocksDB, LLVM). This selection covers diverse application domains, such as security (OpenSSL), blockchain (Bitcoin), database management (RocksDB), and compiler infrastructure (LLVM). This diversity ensures that ComBench accurately reflects a variety of real-world development scenarios.

Table 6: Overview of Projects Analyzed in COMBENCH.

Repository	Language	LoC	Files	Commits	Workflow Runs	Compilation Errors	Avg. Dependent Files per Error
Bitcoin (Bitcoin Core developers, 2009)	C++	376,201	1452	46,307	11,388	12	47
OpenSSL (The OpenSSL Project, 1998)	C	217,931	3809	37,991	100,788	19	13
RocksDB (Facebook, 2012)	C++	724,296	1314	13,420	4706	64	24
LLVM (The LLVM Project, 2003)	C++	99,585	62,932	551,028	1,251,657	105	366
Total		**1,418,014**	**69,507**	**648,746**	**1,368,539**	**200**	**204**



(a) Error Distribution Across ALL Types

(b) Error Type Distribution Across ALL Projects

Figure 4: Comparison of Error Distributions Across Projects and Error Types.

M.2 ERROR TYPE DISTRIBUTION

Following the project characterization, we further analyzed the type distribution of all compilation errors collected across the four projects in our COMBENCH benchmark. This characterization is crucial for two reasons: first, it demonstrates the breadth of error types covered by our benchmark, and second, it provides insights into the typical distribution of compilation errors encountered in real-world C/C++ development.

Figure 4a illustrates the distribution of all 200 compilation errors across the identified categories. Our analysis reveals that declaration errors are the most prevalent, accounting for 85 instances. Following these, type errors (41), member errors (25), and syntax errors (23) represent the next most common categories. Less frequent, but still present, are function errors (19), template errors (4), and semantic errors (3). In total, we identified all seven main error categories within the benchmark, thereby demonstrating comprehensive coverage of common compilation issues. We observe an error type skew, which is consistent with the natural frequency of compiler errors in real-world CI/CD pipelines. While this makes COMBENCH more realistic, it may cause aggregate scores to be dominated by frequent categories. Accordingly, we report both overall and per-type results.

A cross-project analysis, detailed in Figure 4b, further reveals a consistent pattern in the distribution of error types across projects. Across the four analyzed projects, the relative prevalence of error categories generally follows the overall trend: declaration errors are most frequent, followed by type errors, member errors, syntax errors, function errors, template errors, and semantic errors. This consistency suggests that these error types represent fundamental and pervasive compilation issues in C/C++ development, with their relative frequencies being largely independent of specific project characteristics.

M.3 DATA ENTRY COLLECTED AT EACH PHASE

Table 7 provides an overview of the data collected at each distinct phase for every project within COMBENCH. These phases track the data from the initial *Raw Log for Failed Job*, through *Extracted Compilation Error*, *Repair Patch Identification*, and *Deduplication*, culminating in *Success Reproduction*. As evident from the table, the number of entries generally decreases at each subsequent phase, reflecting the progressive filtering and refinement of the data. For instance, LLVM contributes the largest volume of initial raw logs (5,004) and extracted compilation errors (4,412), indicating its complexity and the richness of its error data. This high volume is then significantly reduced through subsequent stages, including deduplication, to yield 105 successful reproductions. In con-

Table 7: Overview of Data Entry Collected at Each Phase for Each Project.

Project	Raw Log for Failed Job	Extracted Compilation Error	Repair Patch Identification	Dedup.	Success Reproduction
Bitcoin (Bitcoin Core developers, 2009)	585	121	76	42	12
OpenSSL (The OpenSSL Project, 1998)	3938	162	27	19	19
RocksDB (Facebook, 2012)	2240	265	146	86	64
LLVM (The LLVM Project, 2003)	5004	4412	168	115	105
Total	**11,767**	**4,960**	**417**	**262**	**200**

trast, Bitcoin, starting with 585 raw logs, undergoes similar filtering, resulting in 42 deduplicated entries and ultimately 12 successful reproductions, suggesting a higher difficulty in achieving full reproduction for its issues. This detailed breakdown highlights the varying data availability and processing challenges across different projects.