Understanding Agent-Based Patching of Compiler Missed Optimizations

Batu Guan
The Chinese University of Hong Kong
Hong Kong SAR
btguan@cse.cuhk.edu.hk

Zirui Wang
The Chinese University of Hong Kong
Hong Kong SAR
zrwang@cse.cuhk.edu.hk

Shaohua Li
The Chinese University of Hong Kong
Hong Kong SAR
shaohuali@cse.cuhk.edu.hk

***Abstract*—Compiler missed optimizations refer to cases in which compilers failed to optimize certain code. It takes many compiler developers' efforts to implement or patch such missed optimizations. In this paper, we present a systematic study of how well agents patch compiler missed optimizations. We identify a significant challenge that patching a missed optimization requires more than just fixing the reported case, and instead requires generalizing to similar cases. We construct a benchmark of real-world LLVM missed optimization issues and compare agent-generated patches with patches from developers in terms of optimization scope. Our results show that coding agents often optimize the given examples, but many generated patches either cover only part of the developer-intended scope or partially overlap with it; in some cases, they further generalize beyond the reference patch. We further introduce historical-knowledge augmentation techniques that leverage prior LLVM optimization pull requests through retrieval and distillation, showing that they improve developer-aligned generalization and yield practical benefits when applied to real-world IR.**

***Index Terms*—compiler missed optimization, llvm, coding agents, generalization**

I. Introduction

Compiler optimization is a fundamental component of modern compiler design and software performance engineering. It aims to improve the efficiency of generated code without altering the intended behavior of the program [1]. Modern compilers, such as LLVM [2], often perform optimizations by identifying specific code patterns and replacing them with more efficient equivalents. This process often requires compiler optimization passes to enumerate and match a large number of code patterns in order to identify potential optimization opportunities in the target code.

However, during the design and implementation of compiler optimization rules, many optimization opportunities may remain unrecognized or unexploited by compiler developers, leading to the problem of missed optimizations. Such missed optimizations can result in suboptimal generated code and have long been a persistent challenge in compiler development [3]. To uncover these missed opportunities, researchers and compiler developers have explored a broad spectrum of techniques, including superoptimizers represented by Souper [4], synthesis-based systems exemplified by Hydra [5], and more recent AI-driven approaches, such as the method proposed by Italiano and Cummins [6] and LPO [3]. As these techniques continue to expose large numbers of missed optimization cases, the main bottleneck is increasingly shifting from identifying optimization opportunities to integrating them as high-quality patches within the compiler maintenance process.

This shift makes automated patching attractive. In particular, recent advances in agents powered by large language models (LLMs) suggest a promising path toward automatically resolving compiler issues through patch generation, thereby reducing substantial manual engineering effort [7]. These efforts primarily demonstrate the potential of LLM agents in conventional compiler issue resolution, particularly in bug fixing. However, missed optimization patching is fundamentally different from conventional bug fixing. For a bug fix, correctness is often approximated by whether the patch satisfies the existing test suite [8]–[10]. By comparison, a missed optimization raises a different challenge that the goal is not merely to make the motivating test cases pass, but to implement the optimization in a way that generalizes appropriately beyond those cases. For example, a reported missed optimization may expose only a concrete instance of a more general rewrite:

$$\begin{aligned}&\texttt{add}(\texttt{umax}(a,10),-10)\Rightarrow\texttt{usub.sat}(a,10)\\&\quad\rightsquigarrow\ \texttt{add}(\texttt{umax}(X,C),-C)\Rightarrow\texttt{usub.sat}(X,C).\end{aligned}\tag{1}$$

A less-general patch may match only the concrete instance, including the particular constant 10. A generalized patch should instead capture the parameterized rule in Eq. 1, which applies across valid choices of X and C. Thus, two patches may both optimize the reported test case while differing in whether they capture the developer-intended optimization scope.

This difference raises a central challenge: generalization. In practice, developers often address a missed-optimization issue through a generalization-oriented workflow. They first observe that a particular intermediate representation (IR) fragment admits a more optimized form, while the current compiler fails to perform the corresponding transformation. Starting from this concrete example, developers then infer and generalize the underlying optimization pattern so that it applies to a broader class of related IR cases, rather than only to the single instance initially reported. Finally, they modify the compiler implementation according to the generalized pattern, producing a patch that can resolve this class of optimization opportunities. We therefore define generalization as the process of transforming a specific, ungeneralized optimization into a more general optimization rule that applies not only to the original code

fragment but also to similar code scenarios, while preserving the correctness of the optimization. Consequently, an agent-generated patch that passes the given tests may still fail to match developer intent by being too narrow, too broad, or misaligned with the intended optimization pattern.

In this paper, we study whether agents can align their generalization with developer intent when repairing missed-optimization issues via patch generation. Rather than focusing solely on whether agents can generate patches that fix the reported instance, we investigate how task context provided to the agents affects their ability to infer the intended optimization pattern and produce patches with developer-aligned generalization. Guided by this benchmark, we structure our study around the following research questions (RQs):

- **RQ1:** How well do agents repair real-world missed-optimization issues, and to what extent do their patches generalize in alignment with developer intent?
- **RQ2:** Does generic generalization instruction improve the alignment between agent-generated patches and developer-intended generalization scope?
- **RQ3:** Can historical-knowledge augmentation help agents infer developer-intended generalization scope for missed optimization?
- **RQ4:** Do the alignment improvements enabled by historical knowledge translate into changes in optimization hit behavior on real-world software?

Our first two studies show that agents do not reliably align patches with developer intent because they often fail to infer the intended generalization scope, and a generic generalization instruction provides limited benefit and can even worsen misalignment. These results motivate a further question: *Can historical LLVM optimization experience provide a useful generalization context for the agent?*

To answer this question, we collect historical LLVM optimization pull requests (PRs) and extract their patches, added regression tests, and summarized generalization behavior. We propose two historical-knowledge augmentation techniques that inject such experience into agents: *retrieval-augmented generation (RAG)* [11], which provides similar source-to-target IR transformations as concrete precedents, and *distillation*, which summarizes recurring component-level generalization principles as compact background knowledge. Our results show that these techniques improve alignment with developer intent across models and yield practical benefits on real-world software, suggesting that effective missed-optimization patching requires reusable developer experience rather than only stronger generalization instructions.

In summary, this paper makes the following contributions:

- We present an empirical study of agent-based LLVM missed-optimization patching; we formulate it as a generalization-sensitive patching problem where success requires matching developer-intended optimization scope.
- We construct a benchmark of real-world LLVM missed-optimization issues with patches from developers and regression tests for scope-based evaluation.
- We use this benchmark to evaluate coding agents under different generalization contexts, showing that agents frequently misalign with developer intent and that generic generalization instructions provide limited benefit.
- We propose historical-knowledge augmentation techniques from prior LLVM optimization PRs via retrieval and distillation, improving intent alignment and real-world IR optimization.

II. Problem Definition

A. Missed Optimization and Generalization

A missed optimization occurs when a compiler preserves program semantics but fails to produce a more efficient form. In LLVM, such cases typically appear as issue reports with an initial IR test case p_0. The compiler leaves p_0 suboptimal, while a developer expects it to be simplified by some optimization pass. Repairing a missed optimization is not merely about making p_0 optimize correctly. The patch must capture the underlying optimization rule and apply it to other related IR whenever the same transformation is valid. We define generalization as the process of inferring a broader optimization pattern from a case, such that the resulting transformation applies not only to the original case but also to semantically similar code patterns, while preserving correctness.

B. Optimization Scope and Assessment

In this work, we define developer intent as the intended optimization scope that the developer aims to capture when fixing a missed-optimization issue, i.e., the set of programs that the developer expects the patch to optimize while preserving semantics. Since this latent intent is not directly observable, we approximate it using the developer-submitted fix, referred to as the golden patch, and the tests added or modified in the fixing commit, referred to as golden test cases. These golden test cases serve as observable samples of the intended optimization scope and provide a lower-bound approximation.

Each patch induces a partial transformation T over LLVM IR programs. The optimization scope $S(T)$ is the set of programs on which T applies and preserves semantics:

$$S(T) = \{p \mid T(p) \text{ is defined and } T(p) \equiv p\}. \tag{2}$$

For each issue, we compare two patches. The golden patch T_G is the developer's fix, and its scope $G = S(T_G)$. The agent patch T_A is generated by an agent, with scope $A = S(T_A)$.

TABLE I: Observed scope relationships between agent and golden patches based on test-based proxies.

Relationship	Interpretation
$A \subset G$	Agent under-generalizes: misses intended cases.
$A \bowtie G$	Partial overlap: misses some intended cases while adding others outside G.
$G \subset A$	Agent generalizes beyond the golden scope, potentially covering a broader optimization space.
$A \sim G$	No observed difference between agent and golden scope.

Our evaluation asks how the observed behavior of the agent patch relates to the golden patch, i.e., whether the agent captures developer-intended scope as approximated by these proxies, not merely whether it optimizes the initial test case. In practice, exact enumeration of A and G is infeasible. We approximate their relationship using two types of tests, detailed in Section III-C. Golden tests verify that the agent covers cases the golden patch is expected to optimize, while fuzz tests search for cases where the agent optimizes differently from golden patch, especially cases optimized only by the agent.

We acknowledge that this proxy-based assessment cannot guarantee perfect enumeration of A and G. A patch passing all golden tests may still miss some intended cases not covered by the tests, and a patch optimizing a fuzz-generated case outside golden tests might represent a valid broader generalization that developers do not explicitly test. Our classification is therefore an empirical characterization of observed equivalence or divergence, not a formal proof of intent.

Based on these signals, we classify the observed relationship between the behavior of the agent patch and the golden patch into four categories, as shown in Table I. These categories form the basis for our empirical assessment in Section III-C.

III. Study Design

To investigate whether agents can infer generalized optimization patterns from initial test cases and generate patches aligned with developer intent, we first construct a benchmark from real-world LLVM missed-optimization issues, then design an agent harness for controlled patch generation, and finally establish a generalization assessment procedure that compares generated patches with golden patches.

A. Benchmark Construction

We construct our benchmark from real missed-optimization issues in the official LLVM repository [1]. Fig. 1 schematically illustrates the structure of the LLVM repository on GitHub. Starting from LLVM GitHub Issues, we collect closed issues labeled both fixed and missed-optimization. We exclude reports that are marked as duplicate, invalid, or wontfix, as well as reports whose fixes target components outside the scope of this study, such as backend-specific code generation, Clang, MLIR, or LLDB. For each remaining issue, we automatically identify the corresponding developer fix from issue discussions, linked pull requests, and commit references. In rare cases where this association cannot be inferred reliably from repository metadata, we resolve it manually.

For each selected report, we first record the optimization pass to which the missed optimization belongs. We then extract the developer-written changes from the corresponding fixing commit and use them as the golden patch. We also collect LLVM IR tests added or modified by fixing commit and use IR input in these tests as source IR of golden test cases. To obtain the corresponding expected IR, we run each source IR through the compiler built after applying the golden patch, using the same optimization pass recorded for the report. The resulting source–expected IR pairs are used as the golden test cases. In addition, we extract test case from the initial issue report and refer to it as the initial test case, which serves as the missed-optimization example provided to the agent.

We validate each issue before including it in the benchmark. First, we confirm that the pre-fix version exhibits reported missed optimization on the initial test case. Second, after applying the developer fix, we rebuild LLVM and confirm that the regression test passes with the expected optimized output.

The final benchmark contains 43 verified LLVM missed-optimization instances. Although modest in size, these instances are selected under strict validation criteria, requiring a reproducible initial test, an identifiable developer fix, and recoverable golden tests. This design favors high-confidence scope-based evaluation over a larger but noisier benchmark. Table II shows their distribution across LLVM optimization passes. InstCombine accounts for the majority of cases because it is LLVM's central pass for local IR canonicalization and peephole simplification, where many missed optimizations are reported and fixed. This is consistent with compiler testing research findings, where InstCombine is the most buggy component [12]. The remaining cases involve SimplifyCFG, ValueTracking, ConstraintElimination, and InstructionSimplify.

TABLE II: Benchmark instances by LLVM Passes.

Component	#	Component	#
InstCombine	32	SimplifyCFG	4
ValueTracking	3	ConstraintElimination	2
InstructionSimplify	2		

B. Agent Harness

For each issue, we would like to evaluate an agent's ability to correctly implement the missed optimization. To provide a simple and reproducible reference point, we implement a lightweight agent harness for LLVM missed-optimization issues. The harness connects a model to an issue-specific LLVM workspace, a controlled tool interface, and a validation loop. If the agent exits before producing a valid patch, exceeds its interaction budget, or fails final validation, the run is counted as a failure. Otherwise, the validated diff is recorded as an accepted patch.

1) Environment Setup: For a benchmark instance, the harness prepares an LLVM workspace at the parent commit of the developer fix. To facilitate the agent, we also implement a set of tools, as Table III lists, which the agent can use to interact and modify compilers. The agent is given access to issue-specific context, relevant LLVM source files, and LLVM Language Reference Manual [13], but not to the golden patch or golden test cases.

2) Overall Workflow: After environment setup, the agent is instructed to act as an LLVM developer and generate a patch for the reported missed optimization. Starting from the initial test case, it first infers the optimization pass and the functions to patch. The agent then enters a ReAct loop [14] that interleaves root-cause analysis, source inspection, patch synthesis,

[1] https://github.com/llvm/llvm-project/

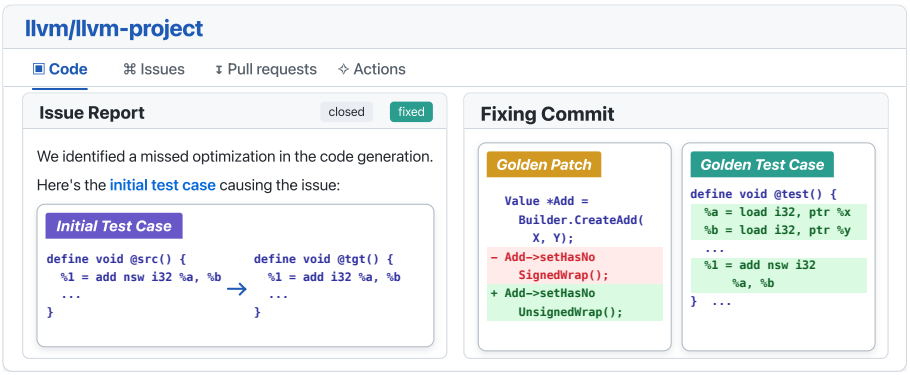


Fig. 1: Illustration of how LLVM repository corresponds to benchmark components.

TABLE III: Tool categories exposed by our agent harness.

Category	Tool	Purpose
Context	`issue_info`, `view_source`, `get_langref`	Inspect tasks, source code and LLVM docs.
Editing	`apply_code`	Apply localized edits to LLVM source files.
Validation	`build_and_check`, `lit_check`	Build, test, and validate candidate optimizations.
Workflow	`show_diff`, `submit_patch`	Review and submit generated patches.

and validation. It uses `issue_info` to inspect benchmark instance, `view_source` to examine LLVM source files, and `get_langref` when the optimization depends on LLVM IR semantics. After identifying a plausible missing optimization rule, the agent applies an edit with `apply_code` and invokes `build_and_check` to validate candidate patch. Validation checks correctness with Alive2 [15] and estimates effectiveness with llvm-mca [16] using block-throughput cost. A patch passes only if its optimized output is no more costly than either original output or golden optimized output. We also run lit [17] to detect regressions, but treat lit failures as soft feedback because FileCheck patterns can be sensitive to non-semantic textual differences. Build errors, lit failures, Alive2 failures, and llvm-mca regressions are returned to the agent for revision. The agent may inspect changes with `show_diff` and submit the final patch with `submit_patch`; the loop ends upon submission or budget exhaustion.

C. Generalization Assessment

We assess the generalization of an agent-generated patch by characterizing its scope relationship with the golden patch, as defined in Section II-B. Given an agent-generated patch A and a human-developed golden patch G, our goal is to determine whether A has the same optimization scope as G, is less general than G, or is more general than G. We accomplish this evaluation using two complementary sources of tests, including golden tests and newly generated fuzz tests.

First, we run golden test cases extracted during benchmark construction on agent-patched compiler. Since these test cases are validated by golden patch, we regard them as representative of the optimization pattern that the golden patch is intended to capture. Therefore, golden test cases are withheld during agent patch generation and used only for post-hoc evaluation, so that they do not bias or interfere with the agent's generalization decisions. If an agent-generated patch fails to optimize these cases, we consider it to exhibit insufficient generalization in the optimization direction intended by the developer.

Second, we fuzz the agent-patched compiler to identify potential divergent or excessive generalization, as illustrated in Figure 2. We use two complementary fuzzing strategies. The traditional fuzzer starts from the initial test case as the seed IR and applies randomized mutations to generate new LLVM IR programs. Inspired by WhiteFox [18], we further design an LLM-based generator to fuzz LLVM IR programs that are likely to expose behavioral differences between the agent-generated patch and the golden patch. Specifically, the LLM-based generator takes the agent-generated patch and the golden patch as input; it is prompted to compare the two patches and synthesize an LLVM IR program that may expose a behavioral difference between them. Each generated program is then optimized by both the golden-patched and agent-patched LLVM compilers. For each candidate program, we first run Alive2 to verify that the optimized IR is a semantically valid refinement of the source, and then check the efficiency of the optimized IR using llvm-mca. If the agent-patched compiler yields a lower llvm-mca computed cost for a program than the golden-patched compiler, we treat it as evidence that the agent-generated patch optimizes cases outside the developer-intended scope.

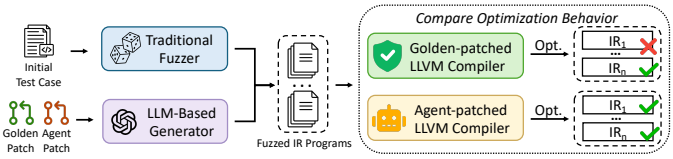


Fig. 2: Fuzz-based generalization assessment. IR_1 will be identified as a program outside the developer-intended scope.

We use these two signals to classify generalization relationship between the agent patch scope A and the golden patch scope G, following Section II-B. Specifically, failing the golden test cases with no agent-only fuzz cases suggests $A \subset G$; failing the golden test cases with agent-only fuzz cases suggests a proper intersection between A and G; passing the golden test cases with agent-only fuzz cases suggests $G \subset A$; and passing the golden test cases with no agent-only fuzz cases suggests likely $A \sim G$, which denotes no observed behavioral difference under our evaluation.

IV. Baseline Study

A. Research Questions

We study whether agents can generate patches whose optimization behavior aligns with the developer intent embodied by corresponding golden patch. We consider two settings. In

the direct setting, agents receive no explicit generalization context. In the context-enhanced setting, agents receive high-level guidance asking for generalization.

- **RQ1:** How well do agents repair real-world missed-optimization issues, and to what extent do their patches generalize in alignment with developer intent?
- **RQ2:** Does generic generalization instruction improve the alignment between agent-generated patches and developer-intended generalization scope?

B. Experimental Setup

We evaluate all agent settings under a controlled setup that fixes the benchmark, tool interface, validation feedback, and interaction budget. This setup isolates the effects of the base model and additional instructions.

1) Models and Agent Framework: We select a variety of both closed and open-source LLMs. Specifically, we evaluate agent performance with four models: GPT-5.5 [19], DeepSeek-V4-Pro [20], Qwen3.5-Plus [21] and Kimi K2.5 [22]. All models use the same harness, issue context, localization hints, tools, and feedback. The agent and the localizer always employ the same model. We use mini-SWE-agent [23] as the coding-agent framework in our experiments.

2) Fuzzing Settings: For the fuzzing component, we use alive-mutate [24] as the traditional fuzzer. Given an input seed program, we generate 1,000 mutants. For the LLM-based generator, considering cost constraints, we use Qwen3.5-Plus. For each patch pair, we ask the model to generate four test cases and repeat this 15 times, totaling 60 test cases.

3) Run Configuration: For each model and each benchmark instance, we run the baseline agent with a fixed interaction budget of 50 steps. Each step corresponds to one ReAct iteration [14], in which the model proposes one or more bash commands and receives the resulting tool output as feedback. A run terminates when the agent explicitly signals submission, exhausts the 50-step budget, exceeds the configured cost limit, or encounters a runtime error. All llvm-mca evaluations are performed under the x86_64 Skylake configuration.

4) Agent Instructions: We evaluate agents under two instruction conditions that differ only in the task-level guidance provided to the agent. In the default condition, the agent is instructed to act as an LLVM developer, diagnose the reported missed optimization, inspect the relevant LLVM source code, implement a minimal fix, and validate the patch using the provided build and test tools. The instructions also specify the available tool interface, the expected edit-test workflow, and constraints such as modifying only LLVM implementation and interface files within the benchmark scope.

To study whether explicit instruction-based generalization context improves alignment with developer-intended optimization scope (RQ2), we add explicit guidance to default instructions: *"Ensure that the final generated patch can handle more general patterns, rather than being limited to the provided test cases."* We intentionally avoid specifying the developer intent in instructions, because our goal is to evaluate whether the agent can, after being encouraged to generalize, infer and align with the intended optimization scope by itself from the issue context and codebase. Apart from this added instruction, the task description, tool interface, validation procedure, and interaction budget remain unchanged.

C. Empirical Findings

We now present the empirical results for the two research questions introduced in Section IV-A. We first examine patch generalization under the default context, and then evaluate the effect of generalization-oriented instructions.

1) RQ1. Generalization Behavior under Default Context: We first evaluate the performance of agents without providing any additional instructions or guidance related to generalization. Table IV reports the performance of agents across different base models. The table reports, for each base model, the number of issues whose agent-generated patch exhibits each relationship with the corresponding golden patch, following the definitions in Section II-B. The Failed column denotes the number of issues for which the agent failed to generate, within the given budget, a patch that optimizes the initial test case.

TABLE IV: Issue counts by patch-scope relationship across base models under the default instruction setting.

Model	Applied					Failed
	$\mathbf{A \subset G}$	$\mathbf{A \bowtie G}$	$\mathbf{G \subset A}$	$\mathbf{A \sim G}$	**Total**	
GPT-5.5	4	4	6	18	32	11
DeepSeek-V4-Pro	5	4	4	14	27	16
Qwen3.5-Plus	9	4	3	11	27	16
Kimi K2.5	11	3	2	10	26	17

From Table IV, we make the following observations. First, in most cases, the agents can successfully generate a patch that can be applied to LLVM and can optimize the initial test case. Specifically, when paired with GPT-5.5, the agent generates such a patch for 74% (32/43) of the issues. This result demonstrates the agents' strong capability in understanding the project context and producing effective patches for the initial test cases. However, among all patches that successfully pass the checks based on the initial test cases, only about half fall into $A \sim G$, where the agent patch matches the optimization scope of the corresponding golden patch. A small number of patches fall into $G \subset A$, as they cover the golden scope and additionally optimize cases beyond it, suggesting potentially broader valid generalization. The remaining patches exhibit scope mismatches. In $A \subset G$, the agent patch covers only part of the golden scope, indicating under-generalization. In $A \bowtie G$, the two patches partially overlap, with the agent patch captures some cases covered by the golden patch but misses others or includes different cases outside the golden scope.

Finding 1. Agents often generate patches that optimize the initial test case, yet they do not necessarily match the golden patch's optimization scope.

2) RQ2. Generic Generalization Instruction : In RQ1, we observed that agent-generated patches do not always match the optimization scope of the corresponding golden patches, even when they optimize the initial test cases. However, under the default instruction setting, the agent is not explicitly

instructed to consider generalization; it is only asked to modify the LLVM source code so that the specified test case can be optimized. Therefore, in this RQ, we investigate whether adding a generic and explicit generalization instruction improves optimization-scope alignment with the golden patch.

After incorporating the generalization instruction described in Section IV-B4, we run the agents again and re-evaluate the generated patches relative to the corresponding golden patches. Table V reports the resulting issue counts by patch-scope relationship across base models.

TABLE V: Issue counts by patch-scope relationship under default and generalization instructions. Gen. Inst. is short for Generalization Instruction. "No" rows are copied from Table IV for easy presentation.

Model	Gen. Inst.	$A \subset G$	$A \bowtie G$	$G \subset A$	$A \sim G$	Failed
GPT-5.5	No	4	4	6	18	11
	Yes	4	2	7	17	13
DeepSeek-V4-Pro	No	5	4	4	14	16
	Yes	8	6	5	10	14
Qwen3.5-Plus	No	9	4	3	11	16
	Yes	10	6	2	8	17
Kimi K2.5	No	11	3	2	10	17
	Yes	5	4	1	9	24

As shown in Table V, adding a generalization instruction does not improve coverage of the developer-intended scope. Because $G \subset A$ is treated as a broader generalization, we consider patches that at least cover the golden scope, i.e., those falling into either $A \sim G$ or $G \subset A$. Surprisingly, the number of such patches does not increase for any base model and even decreases for 3 of the 4 models.

Meanwhile, the number of issues where the agent-generated patch has a narrower optimization scope than the golden patch, $A \subset G$, increases for DeepSeek-V4-Pro and Qwen3.5-Plus. For GPT-5.5 and Kimi K2.5, although $A \subset G$ decreases, the generalization instruction substantially increases the number of Failed issues, indicating that many generated patches no longer optimize the initial test cases derived from the issue descriptions. These results suggest that explicitly instructing the agent to generalize does not reliably improve coverage of the developer-intended optimization scope. In some cases, it may instead shift failures from scope mismatch to initial-test failure or lead to more under-generalized patches.

Finding 2. Generic generalization instructions do not reliably improve golden-patch scope alignment and may reduce success of initial test cases.

V. Historical-Knowledge-Guided Generalization

In the previous sections, we showed that agents do not consistently generate patches that both optimize the initial test case and match the developer-intended optimization scope. Although agents can often address the concrete optimization example described in an issue report, they frequently fail to infer the broader optimization scope captured by the corresponding golden patch. Moreover, adding a generalization-oriented instruction does not reliably improve scope alignment and can even degrade performance in some cases.

One possible explanation is that the agents lack access to the knowledge that compiler developers implicitly rely on when implementing optimization rules. In practice, compiler developers rarely design optimizations in isolation. Instead, they build upon accumulated experience from previously implemented transformations, existing optimization patterns, and historical fixes within the codebase. Such historical knowledge often provides valuable clues about how a newly discovered optimization opportunity should be generalized.

Motivated by this observation, we investigate whether providing historical knowledge can help agents better align their generated patches with developer intent. Specifically, we augment the agent with optimization-related experience collected from prior developer fixes and evaluate whether this additional knowledge improves the agent's ability to generalize beyond the issue-level examples. Through this study, we aim to answer the following research question:

- **RQ3:** Can historical-knowledge augmentation help agents infer developer-intended generalization scope for missed optimization?

To answer this question, we adopt two complementary historical-knowledge augmentation methods: ***RAG*** and ***distillation***. ***RAG***, based on the retrieval-augmented paradigm [11], provides relevant historical optimization patches as concrete examples, while ***distillation***, following the recent trend of skill design [25], condenses recurring patterns in historical changes into compact high-level rules. Together, they provide case-level and principle-level guidance, respectively. Although many other historical-experience augmentation designs are possible, exhaustively evaluating them is beyond our scope. We therefore focus on these two representative and complementary techniques and next describe how their historical knowledge is constructed and integrated into the agent's patch generation process.

A. Historical-Knowledge Augmentation

1) Historical Knowledge Source: We extract historical instances using a procedure largely consistent with the benchmark-construction process described in Section III-A. Unlike benchmark construction, this study collects instances from GitHub pull requests rather than issues, as pull requests offer a larger pool of developer-implemented optimization changes. We exclude all pull requests that are used in benchmark to avoid data leakage. In total, we collect 869 pull requests as the source of historical optimization instances.

For each pull request, we collect source-code patch and accompanying tests, and summarize optimization's generalization behavior for retrieval. Given the patch and tests, the LLM identifies the key pattern, including the initial concrete case, the additional cases covered by the patch of PR, and the transformation logic connecting them. Since pull requests do not explicitly distinguish the pre-generalization test from additional golden tests, we use a simple heuristic under which the first test case is treated as the initial case, while the remaining tests are treated as additional golden cases. Although this heuristic may not exactly match the developer's test-writing

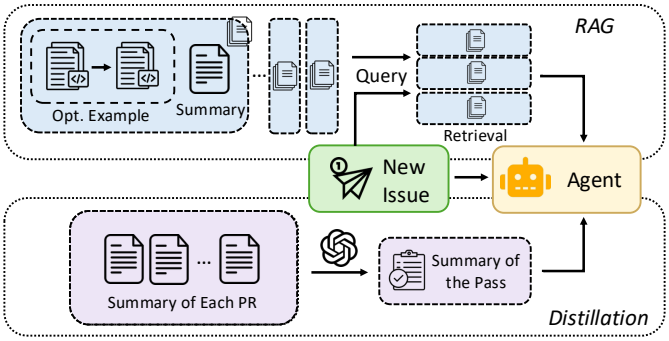


Fig. 3: Workflow of RAG- and distillation-based augmentation.

order, it consistently approximates how the patch generalizes from the original instance to broader coverage.

Overall, each historical instance consists of the source-code patch, the extracted test cases, and an LLM-generated summary of the optimization's generalization details. These instances constitute the historical knowledge source used by both RAG-based augmentation and historical knowledge distillation in the subsequent sections.

2) RAG-Based Augmentation: We use RAG to expose agent to relevant historical optimization experience. The intuition is that RAG provides agent with relevant project-specific examples at patch-generation time, helping it infer intended scope from similar historical patches and test cases. Fig. 3 shows the workflow of RAG-based augmentation. For each historical pull request, we index the optimization behavior reflected by its regression tests. Specifically, each test is represented as a structured source-target IR pair, where the source side denotes the original IR pattern and the target side denotes the expected optimized form. To avoid retrieving examples based on testing syntax rather than optimization semantics, we remove FileCheck directives from the target-side test body before indexing. The resulting source-target pair is serialized using the same representation for both historical documents and online queries, ensuring that retrieval is driven by comparable optimization behavior.

We embed all historical examples under this representation and normalize the resulting vectors for cosine-similarity search. Given a new issue, we encode its initial IR test case in the same format and retrieve the top-k nearest historical examples. To improve diversity, we optionally keep only the highest-scoring example from each pull request.

For each retrieved example, we attach the corresponding historical summary. Thus, the agent receives both concrete behavioral evidence, i.e., the source-to-target IR transformation, and a higher level description of the developer's generalization intent. These retrieved examples are provided as contextual references during patch generation, encouraging the agent to reason by analogy with prior developer-approved optimizations rather than relying solely on the issue-level test case.

3) Distillation-Based Augmentation: In addition to RAG, we distill the generalization experience accumulated across historical optimization fixes into a compact knowledge base and provide it to the agent during patch generation.

The intuition is that historical developer knowledge can be represented at different levels of abstraction. Individual patches provide concrete examples of how a specific missed optimization was repaired, whereas recurring design choices across multiple patches reveal broader generalization practices. For example, developers may generalize an optimization from one predicate to a family of related predicates, from one operand order to both commuted forms, or from a single instruction shape to several semantically equivalent IR encodings. Distillation captures such cross-instance regularities and presents them as explicit background knowledge for the agent.

To construct this knowledge, we aggregate historical generalization summaries by optimization pass and ask an LLM to synthesize pass-level knowledge documents. Fig. 3 shows the workflow of distillation-based augmentation. Each document captures frequently missed elements, such as instruction forms, operand variants, predicates, type constraints, and analysis facts, as well as recurring developer generalization patterns, such as equivalent algebraic forms, multi-instruction canonicalizations, and interactions with value-tracking reasoning. During patch generation, we supply the pass-level distilled document as background guidance for agents.

B. Experimental Setup

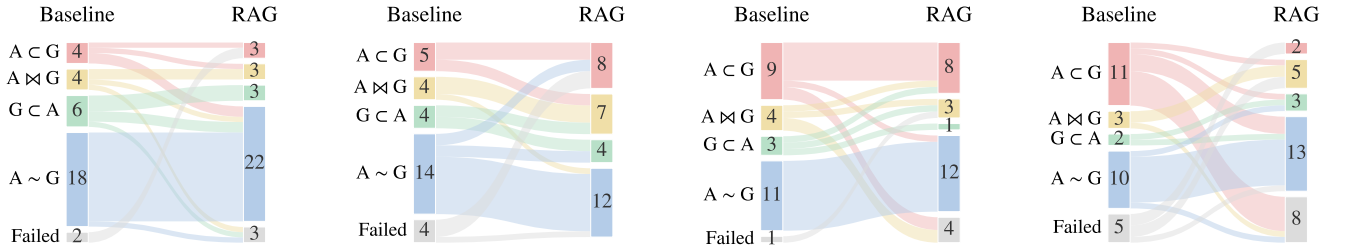
We focus on the experimental settings for the augmentation module. We use DeepSeek-V4-Pro to generate the summary for each historical pull request. DeepSeek-V4-Pro is also used for the one-time knowledge distillation in Section V-A3. This operation is practical because the input consists of compact historical summaries from past pull requests that are distinct from benchmark issues, where each summary is typically a few hundred tokens, and the full set of 869 summaries is well within the 1M-token context window of DeepSeek-V4-Pro. In addition, the distillation is performed offline and grouped by optimization pass, so each synthesis prompt contains only the summaries associated with the corresponding pass.

For RAG, we perform IR-pair matching on a single NVIDIA RTX A6000 GPU. Due to GPU memory constraints, we adopt Qwen3-Embedding-4B [26], a well-known general-purpose embedding model, to compute embeddings for retrieval. For each test case under analysis, we retrieve the top three most similar summaries, i.e., $k = 3$. For distillation, the distilled pass-level knowledge is produced by DeepSeek-V4-Pro using the offline synthesis procedure described above.

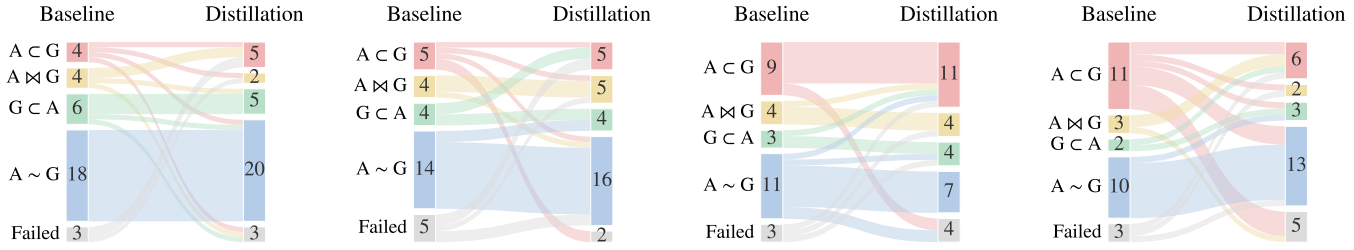
C. Empirical Findings

Fig. 4 shows how the scope relationship between agent-generated patches and golden patches changes after incorporating historical knowledge through RAG-based or distillation-based augmentation. We focus on generated patches that pass compilation and examine whether their optimization scopes exactly match or cover the developer-intended scope represented by the golden patch.

First, we observe consistent gains in exact scope alignment. For every base model, at least one historical-knowledge augmentation strategy increases the number of $A \sim G$



(a) GPT-5.5 (b) DeepSeek-V4-Pro (c) Qwen3.5-Plus (d) Kimi K2.5



(e) GPT-5.5 (f) DeepSeek-V4-Pro (g) Qwen3.5-Plus (h) Kimi K2.5

Fig. 4: Outcome transitions under baseline and different augmentation strategies. Issues whose generated patches fail to compile are classified as Failed. Issues that fail in both compared settings are omitted from the figure.

cases over the non-augmented baseline. For example, GPT-5.5 improves from 18 exact matches to 22 with RAG and 20 with distillation, while Kimi K2.5 improves from 10 to 13 under both augmentation strategies. DeepSeek-V4-Pro also benefits from distillation, increasing from 14 to 16 exact matches, and Qwen3.5-Plus obtains a modest gain under RAG, increasing from 11 to 12. Moreover, the increase in $A \sim G$ is not primarily explained by a reduction in $G \subset A$. For instance, under RAG, GPT-5.5 increases $A \sim G$ by four cases while $G \subset A$ decreases by only three cases; Kimi K2.5 increases $A \sim G$ by three cases while $G \subset A$ also increases from 2 to 3. This suggests that historical knowledge helps agents correct scope decisions that would otherwise lead to misaligned patches, turning more originally non-aligned patches into exact scope matches.

Second, we examine the combined scope-covering outcome $A \sim G \vee G \subset A$, which captures all compilable patches that cover the developer-intended scope. This combined category improves for several base models under at least one form of historical-knowledge augmentation. For GPT-5.5, scope coverage increases from 24 to 25 with RAG. DeepSeek-V4-Pro improves from 18 to 20 with distillation. Kimi K2.5 shows the largest gain, increasing from 12 to 16 under both RAG and distillation. These gains indicate that, beyond increasing exact matches, historical knowledge can also help agents generate patches that cover the developer-intended optimization scope.

Overall, historical-knowledge augmentation mainly improves agents' ability to infer the developer-intended generalization scope by increasing exact scope matches, while also improving broader scope coverage for several models. The gains are model-dependent, but the results consistently show that historical knowledge provides useful guidance for scope inference in missed-optimization repair.

Finding 3. Historical knowledge improves agents' inference of developer-intended scope, increasing exact scope alignment and scope coverage for agents.

VI. Real-World IR Optimization Evaluation

RQ3 indicates that historical knowledge improves agents' ability to decide and adjust optimization scopes, resulting in more exact matches with the golden patch and also more patches that cover the developer-intended scope. We next examine whether these scope-level improvements also translate into greater real-world applicability on downstream IR. We use optimization hits as a proxy for patch effectiveness because they capture whether a patch is exercised on real-world IR, which is a necessary precondition for producing any real-world optimization benefit. Although hits do not directly measure final performance, they provide a scalable and uniform signal across many local LLVM missed-optimization patches.

This study answers the following research question:

- **RQ4:** Do the alignment improvements enabled by historical knowledge translate into changes in optimization hit behavior on real-world software?

A. Experimental Setup

We evaluate the real-world optimization effectiveness of patches by comparing the number of optimization hits they produce on downstream projects. For two patches addressing the same missed-optimization issue, we apply each patch to the same LLVM revision and build two corresponding LLVM compilers. We then use both compilers to process the same real-world project and record how many times the optimization introduced by the patch is triggered.

We count an optimization hit only when the patched optimization actually fires. For patches with multiple modified segments, we instrument the longest segment containing the

core rewrite logic because it typically corresponds to the unique semantic transformation implementing the optimization. Project-level hits sum all firings across the project's IR files, with repeated firings counted independently.

For each base model in the agent, we compare patches generated with historical knowledge against corresponding baseline patches. We use LLVM Opt Benchmark [27] as the source of real-world IR programs in this evaluation. This benchmark provides LLVM IR generated from a broad set of open-source projects. Since the IR is collected from real applications rather than synthetic test cases, it provides a suitable workload for measuring whether generated optimization patches can be exercised in practical compilation scenarios. Due to the high cost of running all benchmark projects for every patch and model, we select eight representative projects from LLVM Opt Benchmark. The selected projects span C, C++, and Rust, and each project has more than 10K GitHub stars, indicating that they are widely used and actively recognized by open-source community. They are `git`, `linux`, `ffmpeg`, `opencv`, `llama.cpp`, `z3`, `uv`, and `tree-sitter`.

B. Empirical Findings

For each issue-project pair, we compare the historical-knowledge-augmented patch with the corresponding baseline patch. The augmented patch is counted as a win if it produces more optimization hits, a loss if it produces fewer hits, and a tie otherwise. Fig. 5 shows the accumulated wins and losses, omitting ties. In RAG setting, all four models achieve more wins than losses, suggesting that retrieved historical cases help patches trigger on more real-world IR patterns. Distillation also shows a positive trend, as three of the four models obtain more wins than losses, with GPT-5.5 showing the largest margin. Overall, historical knowledge improves the practical exposure of agent-generated optimizations on real-world IR, though the effect varies by model and augmentation strategy.

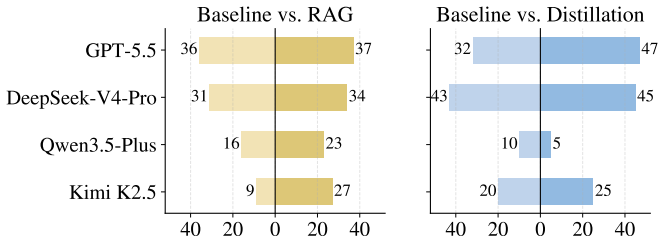


Fig. 5: Accumulated wins and losses of augmented patches against baseline patches over all evaluated issue-project pairs.

Fig. 6 reports project-level cumulative hits aggregated across all models and issues. Distillation improves over the baseline on every evaluated project, increasing total hits from 20.2M to 24.3M across large codebases, such as `opencv`, `uv`, and `linux`, and smaller projects such as `llama.cpp`. RAG yields a smaller and more project-dependent gain, increasing total hits to 20.3M overall, with improvements on projects such as `linux`, `uv`, and `tree-sitter`. These results show that historical augmentation can increase realized optimization

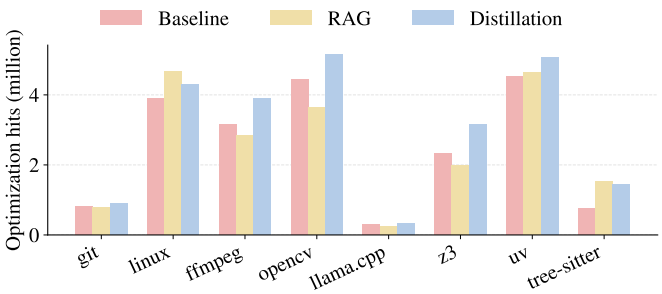


Fig. 6: Project-level cumulative optimization hits of baseline, RAG-augmented, and distillation-augmented patches across all evaluated issues and models.

opportunities on real-world IR, but its effect depends on how the generated patches generalize to each project's IR patterns.

Finding 4. Historical knowledge increases optimization-hit coverage on real-world IR, providing evidence that the generated patches are exercised more frequently in practice.

VII. Case Study

We use LLVM issue #158326 as an example. As shown in Fig. 7, the issue reports a range check combined with a masked equality check: `(a & -2) == 2` restricts `a` to $\{2, 3\}$, making `a < 5` redundant. More generally, the intended pattern is:

$$(X \ pred \ N) \ op \ ((X \ \& \ -2^k) = C), \quad op \in \{\wedge, \vee\}. \tag{3}$$

A direct patch can pass the issue-level test, but the golden patch targets a broader range-reasoning abstraction by converting both ordinary comparisons and masked equalities into `ConstantRange` objects and combining them through intersection or union in `foldAndOrOfICmpsUsingRanges`.

With DeepSeek-V4-Pro, both direct-repair patch and generic generalization instruction-guided patch pass initial test but fail 3 of 5 golden tests. Fig. 8 shows one missed golden case: `(a & -2) == 2` gives `a` $\in \{2, 3\}$, while `a < 3` gives `a` $\in \{0, 1, 2\}$; their intersection is $\{2\}$, so the conjunction should simplify to `a == 2`. The direct patch does not perform this range intersection. Moreover, generic generalization instruction-guided patch expands syntactically to related masks, as illustrated by LLM-generated test in Fig. 9, but still fails to learn the intended semantic range abstraction.

The retrieved examples provide evidence for why the RAG run behaves differently. As shown in Fig. 10, the retrieved context does not reveal the target patch directly. Instead, it exposes a related InstCombine pattern where comparisons over values masked by a negated power-of-two mask can be understood as range constraints over the original value. This context is relevant to the issue because the failing program also combines a range check with a masked comparison.

Following retrieved clue, the agent lifts issue-specific implication into a range-level formulation. As shown in Fig. 10, it treats masked equality as a constraint over the original variable rather than as a purely syntactic pattern. For example, `(x & -2) == 2` denotes the range $[2, 4)$, which is contained in the range required by `x < 5`. This reasoning points the agent

```
define i1 @src(i8 %a) {
  %cmp1 = icmp ult i8 %a, 5
  %masked = and i8 %a, -2
  %cmp2 = icmp eq i8 %masked, 2
  %and = and i1 %cmp1, %cmp2
  ret i1 %and
}

define i1 @tgt(i8 %a) {
  %masked = and i8 %a, -2
  %cmp2 = icmp eq i8 %masked, 2
  ret i1 %cmp2
}
```

Fig. 7: The issue reports a masked equality check implies a range check.

```
define i1 @src(i8 %a) {
  %cmp1 = icmp ult i8 %a, 3
  %masked = and i8 %a, -2
  %cmp2 = icmp eq i8 %masked, 2
  %and = and i1 %cmp1, %cmp2
  ret i1 %and
}

define i1 @tgt(i8 %a)
{
  %and = icmp eq i8 %a, 2
  ret i1 %and
}
```

Fig. 8: A golden test missed by the direct-repair patch.

```
define i1 @src(i8 %a) {
  %cmp1 = icmp ult i8 %a, 5
  %masked = and i8 %a, -3
  %cmp2 = icmp eq i8 %masked, 1
  %and = and i1 %cmp1, %cmp2
  ret i1 %and
}

define i1 @tgt(i8 %a) {
  %masked = and i8 %a, -3
  %cmp2 = icmp eq i8 %masked, 1
  ret i1 %cmp2
}
```

Fig. 9: An LLM-generated test that golden patch does not handle.

1. Retrieved context

```
define i1 @src(i32 %x) {
  %and = and i32 %x, -8
  %cmp = icmp ult i32 %and, 1
  ret i1 %cmp
}
define i1 @tgt(i32 %x) {
  %cmp = icmp ult i32 %x, 8
  ret i1 %cmp
}
```

The key idea is to reason about the value range represented by the masked expression.

2. Agent reasoning

```
icmp ult x, 5          -> x in [0, 5)
icmp eq (x & -2), 2 -> x in [2, 4)
```

Since [2, 4) lies inside [0, 5), the range check is redundant.
Extend `foldAndOrOfICmpsUsingRanges` to:
look through `(x & -Pow2)`,
derive the masked range,
merge it with the icmp range.

3. Patch abstraction

```
if (icmp is equality over
    (X & -Pow2)) {
  maskedRange =
    getRangeForMaskedICmp(...);
} else {
  icmpRange =
    makeExactICmpRegion(...);
}
range=merge(maskedRange,icmpRange);
foldByRange(range);
```

Combine ranges by intersection or union.

Fig. 10: How RAG guides the agent from retrieved masked-comparison knowledge to range-based patch generation. The retrieved example suggests a range interpretation for masked comparisons; the agent applies this abstraction to the issue-level pattern and integrates it into the existing range-combination logic.

to `foldAndOrOfICmpsUsingRanges`, existing InstCombine logic for combining comparisons through ConstantRange. The RAG patch extends range-folding logic by converting masked equality over `x & -Pow2` into a ConstantRange over `x`, allowing existing range-combination logic to simplify the operation, as shown in Fig. 10.

This case illustrates our central finding. The issue contains a masked-equality pattern implying a range check, while direct fixes and generic generalization miss broader predicate combinations. RAG retrieves a related masked-comparison example, prompting range-based reasoning. The resulting patch passes all golden tests, yields no accepted counterexamples, and is therefore considered as behaviorally indistinguishable.

VIII. Related Work

Agents for Issue Fixing. Recent work has explored coding agents for patch generation across diverse repositories [23], [28]–[30], with increasing attention to compiler repair. Zheng et al. [7] propose llvm-autofix, an LLVM-focused agentic harness for bug repair. In contrast, our work targets missed optimizations and their distinctive generalization challenges.

Generalizing Optimization. Generalization has long been studied in compiler. Prior work on peephole-optimization generalization [31], [32] explored symbolic enumeration and proof-centric methods, while Hydra [5] and LPG [33] improve scalability through generate-and-verify workflows and LLM-assisted generalization from real-world optimizations. In contrast, our work unifies generalization and patch generation by starting from an issue report, inferring the intended generalization, and directly producing the corresponding patch.

Developer-Aligned Patch. Prior work distinguishes plausible patches that pass tests from correct patches that reflect developer intent, showing that test-passing patches can still overfit or be incorrect [9], [34], [35]. This has motivated using historical fixes as references for intended behavior [10], [36]–[38]. We extend this idea to compiler missed optimizations, where correctness means matching the developer-intended optimization scope rather than merely passing tests.

IX. Discussion

Potential pretraining-data leakage. Since LLVM patches are public, base models may have seen some benchmark fixes during training. However, generated patches are usually not textually close to the golden patches and often differ in implementation strategy or code location, suggesting that successes are unlikely to be simple memorization.

Stochasticity and evaluation design. We use an open and controllable agent harness rather than productized coding agents with hidden prompts, tools, policies, or updates, ensuring a fixed interaction protocol across all experiments. While agentic repair exhibits stochasticity due to LLM sampling and tool-use ordering, we evaluate each issue–configuration pair under a fixed execution budget and record a single representative outcome to enable consistent issue-level comparison and transition-based analysis. This design focuses on measuring systematic shifts induced by different augmentation strategies rather than run-to-run variance. The observed trends are consistent across four base models and two augmentation strategies and are further corroborated by real-world results.

X. Conclusion

We studied agent-based patching for LLVM missed-optimization issues, focusing on whether generated patches match developers' intended generalization scope. Results show agents often fix reported examples but under- or over-generalize relative to golden patches. Historical knowledge from prior LLVM optimization PRs, via retrieval or distillation, improves scope alignment and can increase real-world IR optimization hits. Overall, effective automated compiler patching requires domain-specific historical knowledge to turn concrete reports into semantically well-scoped patches.

Data Availability

Our research artifacts are publicly available at: https://github.com/cuhk-s3/understanding-generalization.

References

[1] V. Alfred, S. Monica, S. Ravi, U. Jeffrey D *et al.*, *Compilers Principles, Techniques*. Pearson, 2007.

[2] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *International symposium on code generation and optimization, 2004. CGO 2004.* IEEE, 2004, pp. 75–86.

[3] Z. Xu, H. Xu, Y. Tian, X. Zhou, and C. Sun, "Lpo: Discovering missed peephole optimizations with large language models," in *Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ser. ASPLOS '26. New York, NY, USA: Association for Computing Machinery, 2026, p. 1136–1150. [Online]. Available: https://doi.org/10.1145/3779212.3790184

[4] R. Sasnauskas, Y. Chen, P. Collingbourne, J. Ketema, G. Lup, J. Taneja, and J. Regehr, "Souper: A synthesizing superoptimizer," *arXiv preprint arXiv:1711.04422*, 2017.

[5] M. Mukherjee and J. Regehr, "Hydra: Generalizing peephole optimizations with program synthesis," *Proceedings of the ACM on Programming Languages*, vol. 8, no. OOPSLA1, pp. 725–753, 2024.

[6] D. Italiano and C. Cummins, "Finding missed code size optimizations in compilers using large language models," in *Proceedings of the 34th ACM SIGPLAN International Conference on Compiler Construction*, 2025, pp. 81–91.

[7] Y. Zheng, C. Li, S. Li, Y. Zhang, and Z. Su, "Agentic harness for real-world compilers," *arXiv preprint arXiv:2603.20075*, 2026.

[8] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest, "Automatically finding patches using genetic programming," in *2009 IEEE 31st International Conference on Software Engineering*. IEEE, 2009, pp. 364–374.

[9] E. K. Smith, E. T. Barr, C. Le Goues, and Y. Brun, "Is the cure worse than the disease? overfitting in automated program repair," in *Proceedings of the 2015 10th joint meeting on foundations of software engineering*, 2015, pp. 532–543.

[10] X. B. D. Le, D. Lo, and C. Le Goues, "History driven program repair," in *2016 IEEE 23rd international conference on software analysis, evolution, and reengineering (SANER)*, vol. 1. IEEE, 2016, pp. 213–224.

[11] P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Küttler, M. Lewis, W.-t. Yih, T. Rocktäschel *et al.*, "Retrieval-augmented generation for knowledge-intensive nlp tasks," *Advances in neural information processing systems*, vol. 33, pp. 9459–9474, 2020.

[12] Z. Zhou, Z. Ren, G. Gao, and H. Jiang, "An empirical study of optimization bugs in gcc and llvm," *Journal of Systems and Software*, vol. 174, p. 110884, 2021.

[13] LLVM Project, "Llvm language reference manual," https://llvm.org/docs/LangRef.html, 2026, lLVM 23.0.0git documentation.

[14] S. Yao, J. Zhao, D. Yu, N. Du, I. Shafran, K. Narasimhan, and Y. Cao, "React: Synergizing reasoning and acting in language models," *arXiv preprint arXiv:2210.03629*, 2022.

[15] N. P. Lopes, J. Lee, C.-K. Hur, Z. Liu, and J. Regehr, "Alive2: bounded translation validation for llvm," in *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, 2021, pp. 65–79.

[16] LLVM Project, "llvm-mca - LLVM Machine Code Analyzer," 2026, lLVM 23.0.0git documentation. [Online]. Available: https://llvm.org/docs/CommandGuide/llvm-mca.html

[17] ——, "lit - LLVM Integrated Tester," lLVM 23.0.0git documentation. Last updated: 2026-06-12. Accessed: 2026-06-12. [Online]. Available: https://llvm.org/docs/CommandGuide/lit.html

[18] C. Yang, Y. Deng, R. Lu, J. Yao, J. Liu, R. Jabbarvand, and L. Zhang, "Whitefox: White-box compiler fuzzing empowered by large language models," *Proceedings of the ACM on Programming Languages*, vol. 8, no. OOPSLA2, pp. 709–735, 2024.

[19] OpenAI, "GPT-5.5 System Card," https://openai.com/index/gpt-5-5-system-card/, Apr. 2026, updated April 24, 2026. Accessed June 15, 2026.

[20] A. DeepSeek, "Deepseek-v4: Towards highly efficient million-token context intelligence," 2026.

[21] Q. Team, "Qwen3.5: Accelerating productivity with native multimodal agents," February 2026. [Online]. Available: https://qwen.ai/blog?id=qwen3.5

[22] K. Team, T. Bai, Y. Bai, Y. Bao, S. Cai, Y. Cao, Y. Charles, H. Che, C. Chen, G. Chen *et al.*, "Kimi k2. 5: Visual agentic intelligence," *arXiv preprint arXiv:2602.02276*, 2026.

[23] J. Yang, C. E. Jimenez, A. Wettig, K. Lieret, S. Yao, K. R. Narasimhan, and O. Press, "SWE-agent: Agent-computer interfaces enable automated software engineering," in *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024. [Online]. Available: https://arxiv.org/abs/2405.15793

[24] Y. Fan and J. Regehr, "High-throughput, formal-methods-assisted fuzzing for llvm," in *2024 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2024, pp. 349–358.

[25] G. Wang, Y. Xie, Y. Jiang, A. Mandlekar, C. Xiao, Y. Zhu, L. Fan, and A. Anandkumar, "Voyager: An open-ended embodied agent with large language models," *arXiv preprint arXiv:2305.16291*, 2023.

[26] Y. Zhang, M. Li, D. Long, X. Zhang, H. Lin, B. Yang, P. Xie, A. Yang, D. Liu, J. Lin *et al.*, "Qwen3 embedding: Advancing text embedding and reranking through foundation models," *arXiv preprint arXiv:2506.05176*, 2025.

[27] Y. Zheng, "Llvm opt benchmark," 2023. [Online]. Available: https://github.com/dtcxzyw/llvm-opt-benchmark

[28] H. Li, Y. Tang, S. Wang, and W. Guo, "Patchpilot: A cost-efficient software engineering agent with early attempts on formal verification," in *International Conference on Machine Learning*. PMLR, 2025, pp. 35 922–35 941.

[29] Anthropic, "Claude Code by Anthropic — AI Coding Agent, Terminal, IDE," https://claude.com/product/claude-code, 2026, accessed: 2026-05-27.

[30] OpenAI, "Codex — AI Coding Partner from OpenAI," https://openai.com/codex/, 2026, accessed: 2026-05-27.

[31] S. Buchwald, "Optgen: A generator for local optimizations," in *International Conference on Compiler Construction*. Springer, 2015, pp. 171–189.

[32] R. Tate, M. Stepp, and S. Lerner, "Generating compiler optimizations from proofs," *ACM Sigplan Notices*, vol. 45, no. 1, pp. 389–402, 2010.

[33] C. Liao, H. Xu, X. Zhou, Z. Xu, and C. Sun, "Leveraging large language models for generalizing peephole optimizations," *arXiv preprint arXiv:2603.18477*, 2026.

[34] Z. Qi, F. Long, S. Achour, and M. Rinard, "An analysis of patch plausibility and correctness for generate-and-validate patch generation systems," in *Proceedings of the 2015 international symposium on software testing and analysis*, 2015, pp. 24–36.

[35] Y. Xiong, X. Liu, M. Zeng, L. Zhang, and G. Huang, "Identifying patch correctness in test-based program repair," in *Proceedings of the 40th international conference on software engineering*, 2018, pp. 789–799.

[36] D. Kim, J. Nam, J. Song, and S. Kim, "Automatic patch generation learned from human-written patches," in *2013 35th international conference on software engineering (ICSE)*. IEEE, 2013, pp. 802–811.

[37] F. Long and M. Rinard, "Automatic patch generation by learning correct code," in *Proceedings of the 43rd annual ACM SIGPLAN-SIGACT symposium on principles of programming languages*, 2016, pp. 298–312.

[38] J. Bader, A. Scott, M. Pradel, and S. Chandra, "Getafix: Learning to fix bugs automatically," *Proceedings of the ACM on Programming Languages*, vol. 3, no. OOPSLA, pp. 1–27, 2019.