HAFixAgent: History-Aware Automated Program Repair Agent

YU SHI, Queen's University, Canada
HAO LI, Queen's University, Canada
BRAM ADAMS, Queen's University, Canada
AHMED E. HASSAN, Queen's University, Canada

Automated program repair (APR) has recently shifted toward large language models and agent-based systems, yet most systems rely on local snapshot context, overlooking repository history. Prior work shows that repository history helps repair single-line bugs, since the last commit touching the buggy line is often the bug-introducing one. In this paper, we investigate whether repository history can also improve agentic APR systems at scale, especially for complex multi-hunk bugs. We present **HAFixAgent**, a History-Aware Bug-Fixing Agent that injects blame-derived repository heuristics into its repair loop. A preliminary study of all 854 real-world bugs from Defects4J motivates our design, showing that bug-relevant history is both widely available and highly concentrated. Empirical comparison of HAFixAgent with two state-of-the-art baselines shows: (1) Effectiveness: HAFixAgent significantly improves over the agent-based baseline (by 212.3%) and the multi-hunk baseline (by 29.9%). (2) Efficiency: history does not significantly increase agent steps and keeps token costs comparable, with notably lower median costs for complex multi-file-multi-hunk bugs. (3) Practicality: combining different historical heuristics repairs more bugs, offering a clear cost-benefit trade-off. HAFixAgent offers a practical recipe for history-aware agentic APR: ground the agent in version control history, prioritize diff-based historical context, and integrate complementary heuristics when needed.

CCS Concepts: • **Software and its engineering → Software testing and debugging**; • **Computing methodologies → Artificial intelligence**.

Additional Key Words and Phrases: Automated Program Repair, Agentic Software Engineering, Large Language Model, Mining Software Repository, Software Engineering Agent, AI Agent, Bug Fixing

ACM Reference Format:
Yu Shi, Hao Li, Bram Adams, and Ahmed E. Hassan. 2025. HAFixAgent: History-Aware Automated Program Repair Agent. 1, 1 (November 2025), 31 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Software bugs are an inevitable and costly aspect of software development, leading to system failures, security vulnerabilities, and degraded user experience [4, 66, 79]. Since manually identifying and fixing these bugs consumes significant developer time and resources, automated program repair (APR) has been an active and promising research direction for over a decade, aiming to mitigate this burden by automatically generating patches for detected bugs [35]. APR has evolved from heuristic-based [33, 34, 67], constraint-based [32, 44, 48], and template-based techniques [42, 43, 47] to

Authors' Contact Information: Yu Shi, Queen's University, Kingston, Canada, y.shi@queensu.ca; Hao Li, Queen's University, Kingston, Canada, hao.li@queensu.ca; Bram Adams, Queen's University, Kingston, Canada, bram.adams@queensu.ca; Ahmed E. Hassan, Queen's University, Kingston, Canada, hassan@queensu.ca.

learning-based methods [9, 27, 40, 45, 77, 82, 83] and, most recently, large language model-based approaches [16, 24–26, 38, 60, 69, 70, 72, 78, 80] and agent-based approaches [5, 7, 36, 49, 50, 65, 75, 81]. However, despite substantial progress, reliably repairing complex bugs remains a persistent challenge.

Particularly difficult are bugs that span multiple lines and locations (e.g., multi-hunks), which require coordinated, potentially non-local edits across different parts of the codebase. Recent characterizations, such as the work by Nashid et al. [51], highlight that these bugs are not only common but also significantly harder for current LLM-based approaches compared to simpler single-hunk fixes. Overcoming this complexity barrier is crucial for APR approaches to be broadly applicable in real-world software development.

To address such complex bugs, recent advances in APR have pursued two promising yet separate directions. First, agent-based approaches have emerged as the state-of-the-art [5, 65, 75]. Agent-based approaches like RepairAgent [5] leverage LLMs to plan, execute actions via software tools (e.g., code search, editing, testing), and react to iterative feedback, enabling them to tackle more complex reasoning compared to simple prompting. However, these agents typically operate only on the current code snapshot and immediate test results, lacking deeper historical context. Second, drawing from decades of mining software repositories research [1, 21, 63], recent APR techniques started to recognize the value of historical code context. Our prior work [60] showed that injecting historical context derived from git blame[1] commits (the last change touching a buggy line) can improve LLM repair performance for single-line bugs using simple prompting. However, this approach lacked the sophisticated reasoning and tool interaction capabilities of agents, limiting its potential for complex, multi-location repairs.

We hypothesize that combining the iterative, tool-using capabilities of agentic workflows with the rich, contextual clues found in repository history can significantly enhance APR, particularly for complex multi-hunk bugs. History provides information on relevant past changes, developer rationale, and co-evolution patterns, while the agent leverages tools to explore, validate, and integrate this historical context dynamically during the repair process. Yet, it is unclear whether this approach scales. Complex multi-hunk bugs could have many different blame commits, potentially overwhelming the agent with noisy and irrelevant signals rather than providing a focused context.

We design *HAFixAgent*, a history-aware agent for APR that integrates history heuristics (inspired by our prior HAFix work [60]) to enable the agent to leverage historical context alongside bash command tools during its autonomous execution. To evaluate the effectiveness of HAFixAgent, we apply it to all 854 bugs in the Defects4J [29] dataset, a widely used benchmark for evaluating APR techniques. In this study, we address the following research questions (RQs):

- **RQ0. To What Extent Is Historical Information Available in Real-World Bugs?** Before integrating repository history, we have to first establish if relevant historical information is broadly available and scalable for use in APR. Specifically, we analyze the Defects4J dataset to measure the availability and concentration of historical information for bugs. Our findings confirm that 71.1% of bugs have available historical information and most map to a single unique blame commit. This result directly supports the feasibility of using a simple, single-commit heuristic for HAFixAgent, even for complex bugs.
- **RQ1. How Effective Is HAFixAgent at Fixing Real-World Bugs?** We empirically compare HAFixAgent to recent APR systems and also quantify the extent to which any improvements are attributable to the introduction of historical context as opposed to the adoption of an agent architecture. For this, we evaluate HAFixAgent's effectiveness against two state-of-the-art baselines on the Defects4J dataset: one for common bugs [5] and another

[1] https://git-scm.com/docs/git-blame

specializing in multi-hunk repairs [51]. The results show that HAFixAgent repairs 212.3% more bugs than the baseline for common bugs and outperforms the multi-hunk baseline by 29.9%.

- **RQ2. What Are the Cost and Efficiency of HAFixAgent?** Historical grounding should not inflate agent loops or token usage. We track token costs of HAFixAgent under different configurations and compare the costs. We found that integrating history into the agent repair loop is cost-efficient overall, without notable overhead.

The main contributions of our paper are as follows:

- The design and implementation of HAFixAgent, a novel history-aware agentic approach for APR that enriches its decision-making loop with deep repository context.
- A practical and automated method for constructing historical context, featuring three distinct blame-derived historical heuristics and a robust fallback strategy designed for any agentic APR system.
- A large-scale empirical evaluation on all 854 Defects4J bugs, providing strong evidence that history is (1) broadly available (71.1% blameable) and highly concentrated (with 70.7% of bugs mapping to a single commit), and (2) significantly improves agent effectiveness and efficiency.

We will release the complete replication package of code and the evaluation script after the revision is completed. In the interim, please contact us by email to request access.

Paper organization. Section 2 provides background information. Section 3 presents the preliminary study on history availability and distribution. Section 4 describes our architecture design of HAFixAgent. Section 5 presents our case study design. Section 6 presents the results. Section 7 discusses the possible future work. Section 8 outlines threats to the validity of our study. Section 9 reviews related work. Section 10 concludes this paper.

2 Background

In this section, we introduce the background information for our study.

2.1 Mining Repository History for APR

A persistent challenge in LLM-based APR is providing the model with relevant, high-quality context [56, 62]. While many systems focus on the current (buggy) code snapshot, the repository's version control history offers a rich, longitudinal source of context. Decades of Mining Software Repositories (MSR) research have established that historical data is a powerful asset. Hassan [21] demonstrated the broad utility of mining repositories to assist developers. More specifically, Sliwerski et al. [63] showed that analyzing the change-inducing commit can help locate faults. Adams et al. [1] used history to identify co-evolving files or crosscutting concerns, providing a map of code dependencies not visible in a static snapshot.

Building on these MSR insights, researchers have long attempted to create APR systems that directly leverage history. For example, Le et al. [33] pioneered this by creating a repair model based on patterns mined from previous, successful human-generated patches. This idea of using bug-to-fix patterns became a staple in pre-LLM repair techniques. Similarly, Kamei et al. [30] reviewed the state of using historical data for the related task of fixing build failures, confirming the value of this approach.

Even in the modern LLM era, this principle of leveraging history-as-experience is re-emerging to enhance agentic repair. For example, EXPEREPAIR [50] designs a dual-memory system that augments an LLM's context by leveraging historical repair experiences from previously resolved issues from the same repository. Likewise, SWE-Exp [7] proposes an experience-driven framework where the agent learns from the past issue-resolution trajectories (i.e., history) to improve its current issue repair.

These agent-based approaches show the value of history but rely on similar bugs or issues occurring before in the code repository or past trajectories. A more direct and practical heuristic is available through tools like `git blame`. For any given file, `git blame` annotates each line with the specific commit that last modified it. The commit identified by this command serves as a powerful heuristic. This choice aligns with SZZ's core assumption: given a bug-fixing change, SZZ blames the lines modified or deleted by the fix in the pre-fix revision and treats the last change to those lines as a likely bug-introducing commit [31, 63]. Analyzing the patch (diff) and commit message from this single, relevant change can provide critical clues about the code's recent evolution and developer intent. Our prior work, HAFix [60], validated this exact approach, demonstrating that injecting this blame-derived context significantly improves repair for single-line bugs. This paper builds directly on that finding: we aim to explore the hypothesis that integrating this same focused historical context into a modern agentic workflow (Section 2.2) can scale these benefits to handle more complex, multi-location repairs.

2.2 Agentic Software Engineering

The emergence of powerful large language models (LLMs), such as GPT-5 [53] and DeepSeek-V3 [12], has catalyzed a paradigm shift in software engineering, moving from static, one-shot code generation to dynamic, autonomous agent-based systems. An LLM-based agent is a system that couples the core reasoning and planning capabilities of an LLM with essential components like memory (to maintain state) and a set of tools (to interact with an environment) [64]. Unlike a simple prompt-and-response model, an agent operates within an iterative execution loop, often conceptualized as a Reason-Act (ReAct) cycle [76]. This loop allows the agent to formulate a plan, execute an action, observe the outcome, and then autonomously react to that feedback to progress toward a complex, high-level goal.

This iterative, stateful, and interactive approach is the foundation of Agentic Software Engineering (ASE), an emerging field that marks a paradigm shift for software engineering in the era of foundation models [23]. The vision of ASE is to create autonomous "AI Teammates" or "AI Software Developers" that can operate as generalist partners, capable of handling complex, end-to-end software development tasks [22, 39]. The software engineering domain is a particularly fertile ground for agents because developer tasks are goal-oriented (e.g., "fix this bug," "implement this feature") and require interaction with a rich, well-defined tool ecosystem, including file systems, build tools, compilers, test runners, and version control systems [58].

Within the broad field of ASE, a primary and highly active research area has been software debugging and repair, specifically for resolving bugs and GitHub issues. This has led to a new generation of agentic APR systems. For example, SWE-agent [75] introduced an Agent-Computer Interface (ACI) that equips an agent with simple, developer-centric tools (such as file navigation, editing, and searching) to autonomously resolve real-world GitHub issues. Similarly, the OpenHands [65] seeks to build generalist agents that can execute complex tasks by reasoning over a plan and invoking tools. In the specific APR domain, RepairAgent [5] was a pioneering work that provided an LLM with a dedicated toolset to autonomously read and extract code, search code, generate patches, and react to test feedback. Around the same time, systems like AutoCodeRover [81] further enhanced agents by integrating context from program analysis, such as API calls, to improve repair performance.

These systems represent the state-of-the-art in agent-based program repair. However, a common limitation is that the context these agents leverage is typically limited to the current, static snapshot of the codebase and the immediate feedback from the test suite. They generally lack awareness of the codebase's historical and evolutionary context: how and why files have changed over time. HAFixAgent, the APR system proposed in this paper, is designed to bridge this exact gap by enriching this agentic loop with actionable, historical context derived from version control history.

3 Preliminary Study (RQ0): To What Extent Is Historical Information Available in Real-World Bugs?

In this section, we provide the motivation, approach, and results for the preliminary study.

3.1 Motivation

Providing LLM agents with informative context is essential for effective repair [56]. While most context engineering approaches for automated program repair (APR) focus on knowledge obtained from a given snapshot of a code base [8, 15, 20, 38, 56, 70], only recently APR techniques were proposed that leverage code history [14, 60]. Inspired by the domain of mining software repositories [21, 30, 63], our prior work on single-line APR leverages context related to the last commit that touched the bug location, as the latter commit commonly is assumed to be the bug-introducing commit and hence has valuable knowledge about the introduced bug. Finding this commit for single-line bugs is straightforward (as obtained using the `git blame` command), as the latter only comprise one buggy line.

Unfortunately, more complex bugs touching one hunk, multiple hunks in one file or multiple hunks across files potentially would require combining blame commit information of many different lines, potentially drowning the latter information within an LLM's token window, and hence reducing the performance of historical context for APR. In such a case, an APR technique would need to aggregate and rank multiple blame history sources, increasing prompt size, latency, and the risk of distraction. Furthermore, for a given buggy snapshot of the code base, blame commits do not exist for all buggy lines. For instance, when fault localization identifies the fault location as an insertion point (where code is missing) rather than an existing line, `git blame` cannot be applied, as it only annotates existing code. Notably, this process focuses solely on the buggy lines identified by fault localization, which does not depend on the actual bug fix.

This preliminary RQ studies how much of a problem the scale of blame history is for single-hunk, multi-hunk and multi-file bugs: (i) how often does a blame commit exist for buggy lines? and (ii) across how many unique commits is the historical blame information of these types of bugs spread? The results will provide clear empirical insights about the effort required to adapt history-based context to more complex types of bugs.

3.2 Approach

3.2.1 Dataset. To study the availability and distribution of historical information in real-world bugs, we choose the latest version of the Defects4J dataset (v3.0.1) [29], a standard and widely-used benchmark for evaluating recent APR techniques [5, 24, 26, 27, 45, 71, 77]. It contains 854 Java bugs alongside test cases for each bug collected from 17 different open-source projects, spanning various domains such as data visualization (Chart), compiler construction (Closure), and date/time utilities (Time). Analyzing the entire dataset allows us to evaluate the general history distribution on different projects and bugs, without restricting to small or specific bug categories.

Consistent with common practices in the APR field [5, 14, 16, 51, 71, 75, 77], we assume perfect fault localization. Specifically, we use the developer-written patch provided by Defects4J to identify the buggy lines (i.e., the lines in the buggy snapshot that were modified or deleted in the fix). Importantly, our approach does not rely on the actual bug fixes themselves, but solely on the locations of the buggy lines. These identified lines are the only lines on which `git blame` is subsequently computed. We then run `git blame` on these lines to obtain the most recent commits modifying

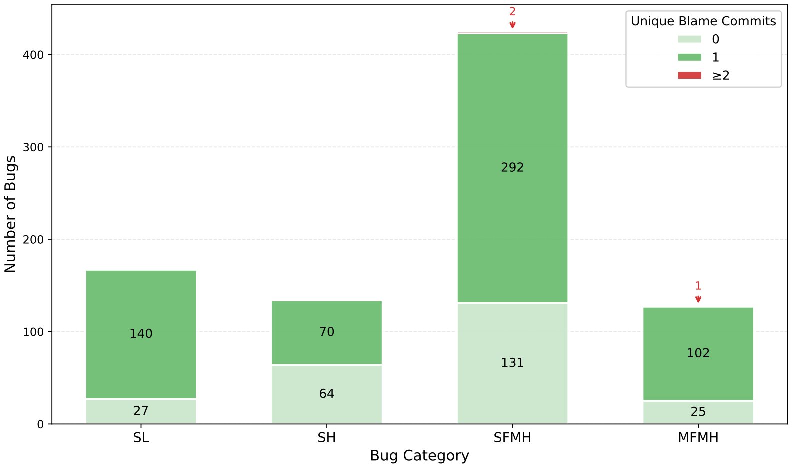


Fig. 1. Distribution of the number of unique blame commits across different bug categories on Defects4J.

those lines. Finally, we de-duplicate the commit hashes obtained for each bug, count each bug's unique blame commits, and compute the distribution by bug category.

3.2.2 Bug Category Terminology. To categorize bugs based on the structure of their bug fix commits, we first define a **hunk** as a single, contiguous block of edits (i.e., additions, deletions, or modifications) applied to a specific region in the source code, similar to earlier work [51]. A patch file can contain one or more such hunks. Furthermore, following prior work on bug categorization [5, 26, 51, 72], we classify all 854 bugs into four mutually exclusive categories based on the number and location of these hunks:

- Single-Line (SL): The bug fix commit edits exactly one line, contained within a single hunk in a single file.
- Single-Hunk (SH): The bug fix commit edits one contiguous hunk in one file, and consists of two or more line changes.
- Single-File-Multi-Hunk (SFMH): The bug fix commit contains two or more distinct hunks, but all hunks are confined to a single file.
- Multi-File-Multi-Hunk (MFMH): The bug fix commit contains hunks that span two or more different files.

We also categorize bugs by blame availability:

- Blameable: At least one edited line in the bug fix commit is a deletion or modification relative to the buggy code snapshot, so `git blame` can return the most recent commit touching such lines.
- Blameless: The bug fix commit only inserts new lines or files, with no deletions or modifications, so no line maps to a prior commit and hence `git blame` is not applicable.

Unlike prior work that either nests SL within SH [26] or collapses SFMH and MFMH into one multi-hunk class [51], we keep all four categories separate for finer granularity.

3.3 Results

Historical blame commit information is broadly available, even for complex bugs. As shown in Table 1, the majority (71.1%) of bugs in Defects4J are blameable, i.e., have at least one blame commit across all buggy lines. Notably,

Table 1. Blame availability by bug category in Defects4J, assuming perfect fault localization. Percentages in the Blameable or Blameless columns are within-category; Total shows the share relative to the full dataset.

Category	Blameable		Blameless		Total	
	#	%	#	%	#	%
SL	140	83.8	27	16.2	167	19.6
SH	70	52.2	64	47.8	134	15.7
SFMH	294	69.2	131	30.8	425	49.8
MFMH	103	80.5	25	19.5	128	15.0
Total	**607**	**71.1**	**247**	**28.9**	**854**	**100.0**

complex multi-hunk bugs (SFMH and MFMH), which account for 64.8% of the entire dataset, maintain high history availability: 69.2% of SFMH bugs and 80.5% of MFMH bugs are blameable. Meanwhile, SL has 140 of 167 blameable (83.8%) and SH has 70 of 134 (52.2%). SFMH is the largest subset at 49.8% of all bugs.

These numbers indicate that historical blame data is commonly available across categories, and hence can be used to ground the reasoning of an APR technique. That said, for the almost thirty percent for which no blame commit can be determined, a workaround would need to be provided.

All but three of the blameable Defects4J bugs have exactly one (!) unique blame commit. Figure 1 shows the distribution of unique blame commits across all bugs. Contrary to expectations, 70.7% (604 out of 854) of all bugs map to exactly one unique blame commit To validate this finding, we manually inspected a sample of multi-hunk cases and confirmed that their blameable lines did indeed trace back to a single, common commit. While one would expect multi-hunk bugs to require at least one separate blame commit per hunk, it turns out that the (blame) history of buggy hunk lines overlaps almost exactly. Those blame commits could still change other lines in addition to the buggy lines (potentially diluting blame context for APR), but the need to add context for only one blame commit suggests that adapting this heuristic to more complex bug categories could be feasible!

In contrast, 28.9% (247 out of 854) have no blame commits (the add-only, blameless cases), and only 0.4% (3 out of 854) of the bugs have two or more git blame commits. The same pattern holds within each category, for example, SL has 140 bugs with one blame commit and 27 with none, and MFMH has 102 bugs with one blame commit and 25 with none.

Summary for Preliminary Study (RQ0):

(1) Blame commits can be identified for 71.1% of bugs (when assuming perfect fault localization).
(2) Even for complex bug types, blame history typically is concentrated into one commit, as 70.7% of Defects4J bugs have exactly one unique blame commit.

4 HAFixAgent

Based on the findings of the preliminary study, we propose a new APR approach called HAFixAgent that exhibits the following three design decisions:

(1) a minimal agent-based architecture designed to iteratively handle complex, multi-step reasoning and actions (e.g., code exploration, patch validation) required for bugs that go beyond simple, single-line fixes.

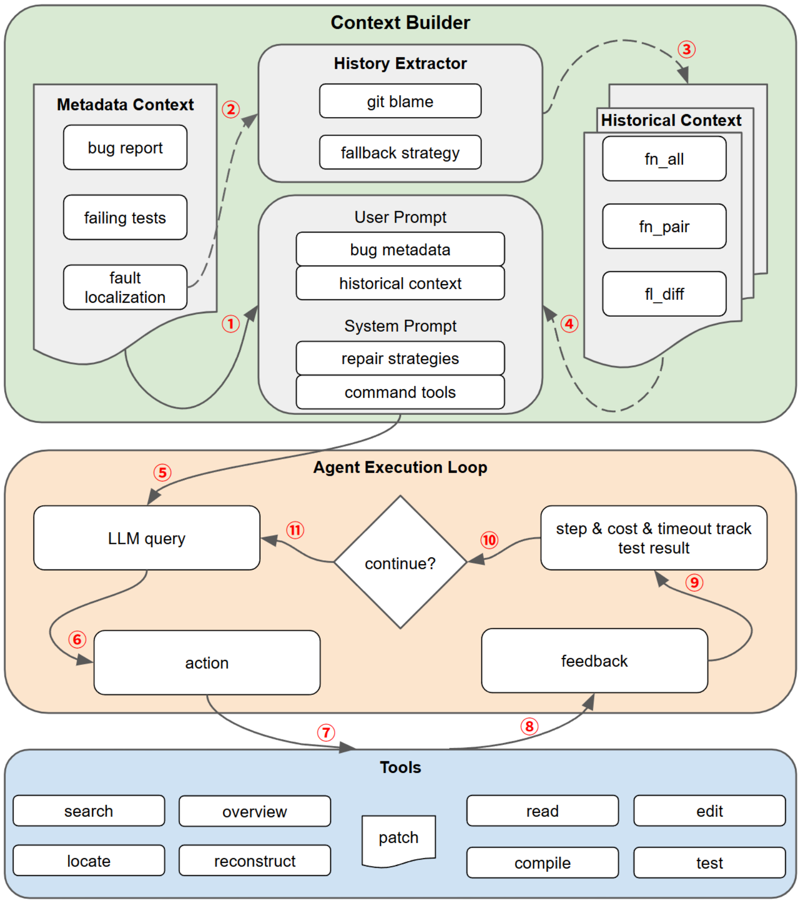


Fig. 2. HAFixAgent architecture and workflow.

(2) the agent has access to a git blame tool, inspired by the findings in Section 3, and leverages the single and unique blame commit as the primary source of historical context.

(3) the agent includes a fallback mechanism to obtain the relevant history for blameless bugs where no buggy line has an associated blame commit.

Unlike agent-based APR systems such as *RepairAgent* [5], which employ pre-defined Python tool APIs, HAFixAgent is intentionally minimal and history-aware. It treats repository history from previous snapshots as an important source of context that augments the current code snapshot, and it operates within a compact, guarded loop over a small set of standard bash tools (e.g., `grep`, `sed`, `find`). This design choice is critical: it minimizes ambiguity from complex and high-level tool interactions, allowing us to more directly attribute performance gains to the quality of the historical context rather than to sophisticated tool exploration. Additionally, the design is consistent with recent commercial coding assistants such as Claude Code [2] and OpenAI Codex [54].

Figure 2 overviews the architecture and its three primary modules. The workflow proceeds as follows:

- **Context Builder**: This module prepares the full context for the agent. It assembles the *Metadata Context* ① and uses its *History Extractor* to retrieve ② and format ③ the *Historical Context*. All context is then organized into the *User Prompt and System Prompt* ④.
- **Agent Execution Loop**: This module controls the iterative repair. It starts with the LLM query ⑤, which generates an action ⑥. This action is sent to the Tools module ⑦. The loop then receives the raw feedback ⑧ from the tools, tracks the test result ⑨, and checks termination guards ⑩ to decide whether to continue. If so, the feedback is routed to the next LLM query ⑪, repeating the cycle.
- **Tools**: This module provides an isolated sandbox environment for action execution. It receives an action (e.g., search, edit, test) from the execution loop ⑦, then returns the resulting feedback (e.g., stdout/stderr) to the loop ⑧.

4.1 Context Builder

4.1.1 Metadata Context. The user prompt contains three items: the human-written bug report, the initial set of failing tests, and the fault localization context, which specifies the buggy files and lines. These metadata are used only as *non-history* context and are rendered in step ①.

4.1.2 History Extractor. The *History Extractor* module (step ② in Figure 2) is responsible for finding a relevant blame commit for every bug. We use the buggy lines identified through the fault localization metadata to obtain the blame commit, since the historical context in that blame commit may reveal the root causes of the bug [21, 30, 63]. The module handles blameable and blameless bugs differently:

- Blameable bugs. For bugs involving modified or deleted lines, we execute `git blame` on all identified lines and collect the union of all unique commit hashes. This process is deterministic rather than a random selection of one line. As demonstrated in our preliminary study (Section 3), this union is nearly always a single commit: 70.7% of all bugs (except three blameable bugs) map to exactly one unique blame commit.
- Blameless bugs (fallback strategy). To enable these blameless bugs (i.e., add-only fixes) to also benefit from historical context, we design a fallback strategy. Specifically, we employ a fallback strategy to blame the nearest executable code line (e.g., an executable line rather than a comment or symbol) within the five lines preceding the insertion. This strategy is based on the assumption that new code is often added near existing, related logic, making the most recent commit to that local area a relevant source of historical context.

4.1.3 Historical Context. Once the blame commits are determined by the *History Extractor*, step ③ then mines relevant historical context from the identified blame commits by leveraging one of the three top-performing historical heuristics from our HAFix approach [60] (which focused on SL bugs only):

- All functions' names from the blame commit (*fn_all*): includes the names of all functions from all co-changed files in the blame commit. This setting captures a broad structural view of the co-evolved functions within the same history commit.
- Historical function before and after (*fn_pair*): which contains the before and after snapshot of the function code body that includes the buggy line. This historical context helps the agent understand how the buggy function evolved and provides possible clues about the root cause of the bug.

- File-level diff (*fl_diff*): includes the diff patch obtained from the git diff command in the blame commit. This configuration exposes the exact code modification across files, allowing the agent to reason about fine-grained edits that may have caused the bug.

In step ④, HAFixAgent integrates the history context, which was extracted in the previous step ③ using one of the three heuristics from the blamed commit. The history is injected directly into the User Prompt (as shown in Figure 2), with truncation safeguards (e.g., truncating an exceptionally large file diff) applied to ensure that the total prompt fits within the LLM's context window. This process leads to our four context configurations: the agent operates either with the *non-history* context only (from Section 4.1.1) or with the *non-history* context augmented by one of the three history heuristics (*fn_all*, *fn_pair*, or *fl_diff*).

4.2 Agent Execution Loop

HAFixAgent's execution loop follows an observe-act-feedback paradigm, inspired by frameworks like ReAct [76] and those used in recent software engineering agents [75]. The loop proceeds with the following steps: ⑤ the LLM receives the current state (including the full conversation history of prior actions and observations); ⑥ the agent parses the model output to extract a tool invocation (i.e., a bash command, which the system prompt constrains the model always to provide) as the next action; ⑦ the command runs in a containerized tool sandbox that exposes repair utilities and dataset tools (e.g., search, edit and compile); ⑧ stdout and stderr are collected from the output of command execution as feedback; ⑨ compilation and test outcomes are captured and parsed; ⑩ the loop terminates on success or when a termination guard (e.g., step limit, cost limit, or timeout) is reached; otherwise, ⑪ the observation is appended to the conversation history and fed back to the LLM for the next reasoning step.

4.3 Tools

Table 2 lists the bash tools that HAFixAgent may invoke during the execution loop. The agent emits exactly one bash command per step, with optional short chaining via `&&` or `||`. These actions are executed between steps ⑦ and ⑧ in the loop, and the same toolset is embedded in the system prompt (Appendix A).

HAFixAgent intentionally exposes a simple but powerful set of bash tools together with project commands like `compile` and `test`. A key design choice is that the historical context is not an interactive tool, but is instead modularized and prepared by the *History Extractor* and injected directly into the agent's context before the execution loop begins. This decouples history retrieval from agent actions, which allows the agent's toolset to remain simple, modular, scalable and focused on repair-related operations (e.g., read, edit and test).

The tools in Table 2 fall into three groups. This specific set of tools was chosen to be minimal yet complete, providing the fundamental capabilities (file operations, project compilation, and testing) required for the iterative repair process while relying on universal, standard Linux utilities. (i) *File operations*: `grep` gives a quick structural overview, `sed -n` reads targeted windows around fault locations with progressive expansion, `sed -i` applies precise edits, `head`/`tail` supports multiline reconstruction by splicing, and `find` locates candidate files. (ii) *Project commands*: `compile` command checks build health after edits; `test` command runs only the relevant tests to provide fast feedback in the execution loop. (iii) *General commands*: a completion sentinel (`echo COMPLETE_REPAIR_SIGNAL`) signals success, and simple chaining helps combine a small read or edit with an immediate compile or test.

In practice, the loop alternates between these actions: inspect and read around the localized lines, apply a minimal edit, compile, run relevant tests, and iterate until all tests pass. Termination occurs in two ways: (1) the agent, after

Table 2. Bash tools used by HAFixAgent.

Tool	Description	Usage Example
Repair Operations		
`grep`	Search for patterns in files; Quick file overview of classes, interfaces, and method signatures	`grep -n "class|interface|public.*(" file.java`
`sed -n`	Read specific line ranges from files; extract targeted context around fault locations progressively	`sed -n ’45,65p’ BuggyClass.java` (reads lines 45-65)
`sed -i`	In-place file editing; make precise, targeted fixes to source code	`sed -i ’s/oldCode/newCode/’ file.java`
`find`	Locate files in the repository by name or pattern	`find . -name "*Test.java"`
`head` / `tail`	Reconstruct files for complex multi-line edits; extract beginning/end of files for rebuilding	`head -n 50 file.java > temp && cat fix.txt » temp`
Project Commands		
`compile`	Compile the project to verify setup and check for compilation errors after edits	`compile`
`test`	Run relevant/failing tests; verify bug fix effectiveness	`test -r`
General Commands		
`echo`	Signal task completion when all tests pass	`echo COMPLETE_REPAIR_SIGNAL`
`&&` / `||`	Chain multiple commands together; execute sequentially or conditionally	`sed -n ’10,20p’ A.java && sed -n ’30,40p’ B.java`

running a successful `test` command, self-terminates by emitting the `echo COMPLETE_REPAIR_SIGNAL`, or (2) the agent execution loop detects that a guard (step, cost, or timeout) has been reached and halts the process.

5 Case Study Design

In this section, we describe the case study design for evaluating HAFixAgent.

5.1 Prompt Context Preparation

Consistent with the preliminary study, we employ the entire 854 bugs of Defects4J to prepare the data needed for evaluation.

Metadata collection. To prepare the *non-history* context as defined in Section 4.1.1, we need bug metadata such as bug reports, failing tests, and fault locations. We mine bug reports from issue tracker links provided by Defects4J. For 18 cases from the *Chart* project without a clear bug issue link, we use the fixing commit message as the bug description, and to prevent potential data leakage, we manually verified these commit messages to ensure they do not contain explicit fix instructions. Similar to the study [51], we collect the failing tests provided by Defects4J. Consistent with common practices in the field of program repair [5, 14, 16, 51, 71, 75, 77], we assume perfect fault localization and provide fault localization context, which is derived from the developer's bug fix (following the process in Section 3.2), directly into the agent's context. Specifically, similar to prior work [5], this context consists of all file(s) and line(s) that need to be edited to fix the bug.

Historical data collection. Motivated by the preliminary study in Section 3, which shows that 71.1% of bugs are blameable and most map to a single unique blame commit, we collect history per bug as follows:

- Blameable with one unique commit: Run `git blame` on deleted or modified lines to obtain the blame commit, as defined in Section 4.1.3.
- Blameable with multiple commits: Only three bugs fall in this case, we use the LLM-as-a-judge (the same LLM as HAFixAgent) to select the single most relevant commit. The model is prompted with the set of blameable lines and asked to choose the line which is most likely related to the bug's root cause.
- Blameless with zero commits: Apply the nearest line fallback strategy from Section 4.1.3 to select a blame commit.

Given the selected blame commit, we extract and construct data of *fn_all*, *fn_pair*, and *fl_diff* as defined in Section 4.1.3.

5.2 Baselines

We compare HAFixAgent with two recent APR techniques as baselines, i.e., RepairAgent [5] and BIRCH-feedback [51].

RepairAgent. RepairAgent is an autonomous LLM-based agent equipped with pre-defined API tooling. The original study evaluated it on 835 bugs (from Defects4J v1.2 and v2), reporting 186 plausible (test-passing) and 164 correct (manual semantic check) fixes. Our comparisons are performed on the 829-bug overlap with our v3 set. For our aggregate effectiveness comparison, we compare our test-passing patches against their reported 186 plausible fixes, as these metrics are equivalent. For our fine-grained, per-category analysis, we are limited to their 164-bug correct set, as bug IDs (which are required for categorization) were only released for this set instead of the plausible bug set.[2]

BIRCH-feedback. BIRCH-feedback targets multi-hunk repairs. We compare results on the 371 overlapping multi-hunk bugs reported in that study and use their best-performing configuration with feedback as the baseline, which we refer to as *BIRCH-feedback.*

5.3 Metrics

In line with previous studies [6, 14, 51, 55, 61], we report *Plausible@1* to evaluate the correctness based on whether the generated patch passes all test cases. Formally:

$$\text{Plausible@1} = \frac{1}{|\mathcal{B}|} \sum_{b \in \mathcal{B}} \text{TestPass}\Big(\text{patch}_1^b\Big) \tag{1}$$

where $\mathcal{B}$ denotes the set of bugs and $\text{TestPass}(\cdot) \in \{0, 1\}$ returns 1 if the patch compiles and all tests pass, and 0 otherwise. We generate one patch per bug using standard decoding settings, with the temperature set to 0.0 for enhancing reproducibility.

[2] https://github.com/sola-st/RepairAgent/issues/18

Complementing *Plausible@1*, which measures the pass rate as a proportion of all bugs), we also report *#Pass* (the absolute count of bugs passing the test suite), similar to prior work [51]. To quantify the unique contribution of diverse historical context, we additionally report *#Unique Pass*, to show the number of bugs solved by a configuration using historical context but not by its *non-history* counterpart. We visualize overlaps using Venn diagrams to illustrate multi-way complementarity.

For efficiency, we report the average number of agent steps per bug, the average cost per bug in USD computed from token usage, and the cost stratified by outcome (successful versus failed fixes). To assess the statistical significance of differences in cost and number of steps, we compare the distributions across the four context configurations. Since the data is paired (same bugs, different configurations) and not assumed to be normally distributed, we employ the Friedman test [18], a non-parametric test for multiple comparisons. If the Friedman test is significant ($p < 0.05$), we conduct pairwise Wilcoxon signed-rank tests [68] for post-hoc analysis to compare each history-aware configuration against the *non-history* configuration. To control for multiple comparisons (three pairs), we apply the Bonferroni correction [13] and set the significance threshold α.

5.4 Implementation

HAFixAgent is implemented in Python and builds on *mini-swe-agent* [75] for the execution loop, which aligns with our lightweight design. While mini-swe-agent (and swe-agent) is designed naturally to support solving PR issue benchmarks such as SWE-Bench [28], it lacks the capability of historical context management (e.g., history localizing and retrieval), the repairing strategies and tools specifically for resolving Defects4J bugs. It is an open-source, lightweight framework with about 100 lines of core loop code, enabling seamless integration of historical context extraction and bash tools orchestration during runtime. We use DeepSeek-V3.2-Exp [11, 12] via the official API with temperature set to 0.0, based on the official DeepSeek recommendation [10]. Prompts are rendered at runtime with *jinja2*.

We run HAFixAgent on Ubuntu 20.04, with each bug executing in an isolated Docker container. This sandbox environment is built from the released Defects4J image. For this evaluation, we implement the abstract `compile` and `test` tools (described in Section 4.3) as wrappers for the `defects4j compile` and `defects4j test -r` commands, respectively. The container is preinstalled with all necessary Defects4J and standard bash tools required by HAFixAgent. For each bug, we cap the agent's loop at 50 steps, enforce a per-bug cost guard of $1 USD, and set a 1-hour timeout. After each run, the container is automatically cleaned, while all generated patches and logs are persisted for analysis.

6 Results

In this section, we provide the motivation, approach, and results for each of our research questions.

6.1 RQ1: How Effective Is HAFixAgent at Fixing Real-World Bugs?

6.1.1 Motivation. Our prior work [60] shows that injecting historical context improves LLM repair for single-line bugs under simple prompting. History carries heuristic signals that are hard to infer from a static snapshot: the fault introducing change, coevolving functions and files, and often intent hints in commit diffs and messages. RQ0's finding that most Defects4J bugs, even multi-hunk ones, have only one single historical blame commit, suggests that historical context could also scale to APR techniques for more complex bug types.

This RQ rigorously validates the benefit of blame-derived context on SH, SFMH and MFMH bugs, in the context of our proposed HAFixAgent. To thoroughly validate our approach, we first establish HAFixAgent's practical viability by benchmarking it against state-of-the-art (SOTA) baselines to determine if our history-aware method is competitive.

Second, we isolate the specific impact of our core hypothesis by evaluating the agent across different context configurations (i.e., "with" versus "without" history). This analysis is essential to pinpoint and quantify the precise contribution of historical context and to understand how these gains vary by bug category. This motivates the following questions: (a) is HAFixAgent competitive against SOTA baselines? (b) what is the performance impact of adding historical context to an APR agent, and (c) how do these gains vary by bug category?

6.1.2 Approach. We compare HAFixAgent against two SOTA baselines on shared bug subsets: *RepairAgent* (829 bugs) and *BIRCH-feedback* (371 multi-hunk bugs). For *RepairAgent*, our direct comparison uses the set of bugs they reported as correctly patched, as their full list of plausible bug IDs is unavailable; we also report their published aggregated counts (Section 5.2) for reference. To analyze performance across different bug types, we categorize results by our four bug types (SL, SH, SFMH, MFMH). We pay special attention to the multi-hunk categories (SFMH and MFMH), as their need for coordinated changes may make them more sensitive to historical context. *BIRCH-feedback* serves as a strong baseline for this specific subset.

In addition, we evaluate the four context configurations defined in Section 4.1 across all 854 Defects4J bugs: HAFixAgent-*non-history*, HAFixAgent-*fn_all*, HAFixAgent-*fn_pair*, and HAFixAgent-*fl_diff*. Notably, 247 bugs (28.9%) are blameless (fix adds only new lines), precluding standard blame. To enable these blameless bugs to also benefit from historical context, we apply the fallback strategy (defined in Section 4.1.3), ensuring every bug has meaningful historical context.

All experiments use the same LLM, identical parameters, and a consistent containerized runtime. We report both Plausible@1 (computed via termination checks in Section 4.2) and #Pass (the total number of bugs with patches passing all tests). To quantify the specific benefit of historical context, we also report #Unique Pass for the number of bugs repaired by a history-aware configuration that were not repaired by the *non-history* configuration.

6.1.3 Results. **HAFixAgent substantially outperforms both RepairAgent and BIRCH-feedback.** On the 829 bugs shared with RepairAgent (Table 3a), the best context configuration of HAFixAgent (*fl_diff*) mode repairs 523 bugs versus 164 correct fixes reported by RepairAgent, a +218.9% overall improvement. Across different context configurations of HAFixAgent, the improvements relative to this 164-fix baseline range from +206.1% to +218.9% (502, 510, 514, 523), with an average of 212.3% improvement. When comparing against RepairAgent's reported aggregate plausible total of 186 (the equivalent metric), the corresponding improvements are +170.0% to +181.2% (175.4% by average).

On the 371 bugs shared with BIRCH-feedback (Table 3b), HAFixAgent also achieves higher totals. The best context configuration (*fn_pair*) repairs 175 bugs versus 133 for BIRCH-feedback, a +31.6% improvement. Across four context configurations, the gains range from +28.6% to +31.6%, with an average of 29.9% improvement. By category, the best HAFixAgent configuration (*fn_pair*) repairs 133 SFMH and 42 MFMH bugs, compared with 100 and 33 for BIRCH-feedback. Overall, HAFixAgent is consistently effective than both SOTA baselines across shared subsets and categories.

We note, however, that these SOTA comparisons are confounded by the use of different LLMs (RepairAgent used GPT-3.5, BIRCH-feedback used o4-mini, and our work uses DeepSeek-V3.2-Exp). Therefore, this analysis primarily establishes HAFixAgent's competitiveness, while our internal ablation study (comparing history to non-history with the same LLM) provides the direct, controlled evidence for our claims.

History configurations fix more unique bugs compared to the *non-history* configuration, with the largest gains for the multi-hunk bug categories. Table 4 summarizes the number of passing and uniquely passing bugs across the four context modes, while Figure 3 visualizes their overlap and complementarity. Across all 854 bugs, the three history-based configurations add 194 bugs that the *non-history* configuration never fixes, whereas the *non-history*

Table 3. Effectiveness comparison of HAFixAgent against RepairAgent and BIRCH-feedback on their shared subsets. For HAFixAgent and BIRCH-feedback, cells report the number of bugs with plausible patches that pass the full test suite. For RepairAgent, we report the number of bugs with correct patches because their repository does not provide plausible bug IDs, and only reports aggregates (plausible = 186, correct = 164). Since correct is a subset of plausible, RepairAgent's counts are more conservative compared to the plausible numbers for BIRCH-feedback.

(a) 829 common bugs with RepairAgent

Category	Common	HAFixAgent				RepairAgent
		non-history	*fn_all*	*fn_pair*	*fl_diff*	
SL	165	135	134	134	138	99
SH	128	93	91	102	98	30
SFMH	408	224	238	235	239	31
MFMH	128	50	47	43	48	4
Total	829	502	510	514	523	164 (186*)

* The 186* denotes the total number of plausible patches of RepairAgent; specific bug IDs are only available for correct patches in their repository.

(b) 371 common multi-hunk bugs with BIRCH-feedback

Category	Common	HAFixAgent				BIRCH-feedback
		non-history	*fn_all*	*fn_pair*	*fl_diff*	
SFMH	244	123	125	133	126	100
MFMH	127	49	46	42	47	33
Total	371	172	171	175	173	133

configuration contributes only 32 unique fixes not recovered by any history configuration. By category, *fl_diff* performs best on SL with 140 total fixes and 11 unique ones and *fn_pair* leads on SH with 108 fixes and 80.6% Plausible@1. In SFMH, *fl_diff* achieves 254 fixes and the three history configurations collectively add 119 more repairs, while the *non-history* configuration uniquely fixes 17. Finally, the MFMH category reveals a complex trade-off. Although the *non-history* configuration achieves the highest total fixes (50), this aggregate score masks a clear complementarity. Collectively, the history configurations add 24 unique fixes not found by the *non-history* agent, outweighing the 8 unique fixes from the *non-history* side (Figure 3). This suggests that while history can sometimes distract the agent on complex multi-file bugs, it also provides an orthogonal signal that unlocks repairs for a different subset of bugs.

The three history configurations are complementary rather than redundant, solving distinct sets of bugs that others miss. The Venn diagrams in Figure 3 show large shared cores in each category (SL 118, SH 72, SFMH 140, MFMH 22), yet each heuristic contributes nontrivial unique fixes, especially in SFMH, where *fn_all*, *fn_pair*, and *fl_diff* add 21, 19, and 19 unique cases, respectively. This pattern shows that *fn_all* captures co-evolution across functions, *fn_pair* captures before and after semantics, and *fl_diff* captures fine-grained edits at the file level, so the union of history-based configurations yields broader repair coverage than any single configuration.

Table 4. Repair performance across different context configurations over 4 bug categories on Defects4J. # Pass represents the number of bugs passing the test suite. Unique # Pass represents the number of bugs solved by that configuration but not by the corresponding *non-history* configuration. The highest value for each category group in each column is highlighted in bold.

Category	Context	# Pass	Plausible@1	# Unique Pass
SL	*non-history*	137	82.0%	-
	fn_all	136	81.4%	10
	fn_pair	136	81.4%	10
	fl_diff	**140**	**83.8%**	**11**
SH	*non-history*	97	72.4%	-
	fn_all	96	71.6%	16
	fn_pair	**108**	**80.6%**	**19**
	fl_diff	103	76.9%	18
SFMH	*non-history*	238	56.0%	-
	fn_all	251	59.1%	67
	fn_pair	249	58.6%	**69**
	fl_diff	**254**	**59.8%**	68
MFMH	*non-history*	**50**	**39.1%**	-
	fn_all	47	36.7%	**13**
	fn_pair	43	33.6%	13
	fl_diff	48	37.5%	12

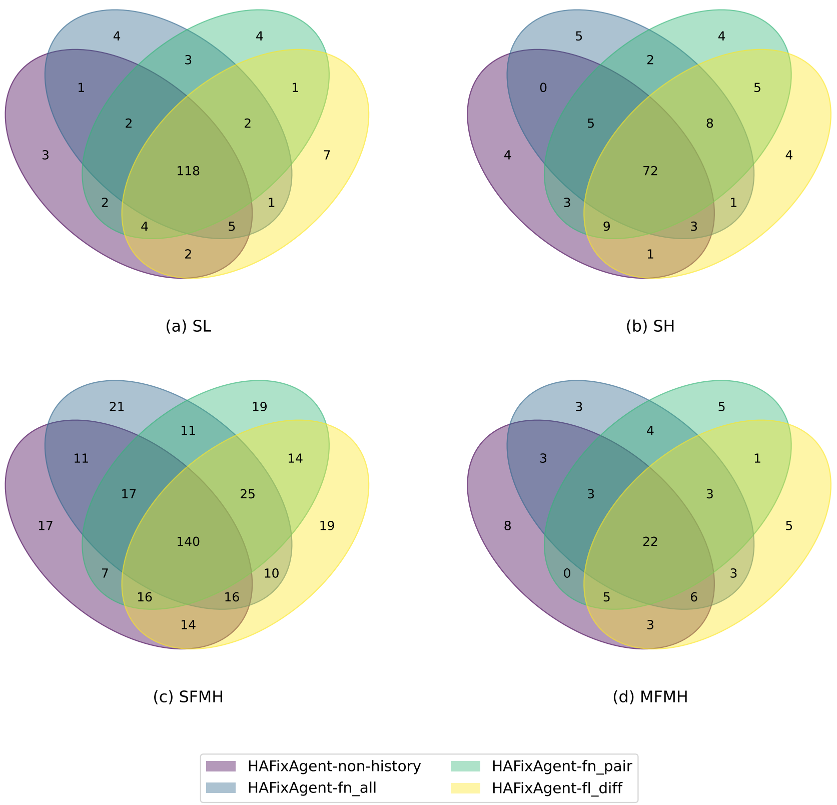
Summary for RQ1:

(1) HAFixAgent outperforms RepairAgent by an average of 212.3% across the different context configurations, and outperforms BIRCH-feedback by an average of 29.9%. Additionally, across all 854 bugs, history configurations add 194 history-only fixes versus 32 non-history-only, with the largest gains in multi-hunk categories.

(2) The three history heuristics solve distinct cases beyond a shared core, indicating that combining heuristics yields broader coverage than any single setting.

6.2 RQ2: What Are the Cost and Efficiency of HAFixAgent?

6.2.1 Motivation. While RQ1 demonstrates that incorporating historical information into HAFixAgent enhances repair performance, it is also important to understand the associated cost and efficiency trade-offs. Cost is a first-order concern for agent systems because iterative tool use can accumulate substantial token and time budgets, even on simple bugs [5, 72]. Recent evaluations also highlight the risk of "token snowballs" and expensive failures (a phenomenon where unresolved attempts consume several times more tokens than successful ones [17], making cost and efficiency as important to measure as accuracy. HAFixAgent integrates additional contextual information into each prompt, which might potentially increase inference cost and agent reasoning steps. Therefore, we analyze whether the performance gains observed in RQ2 come at a significant cost and how efficiently the agent converges toward successful repairs across bug categories.



(a) SL

(b) SH

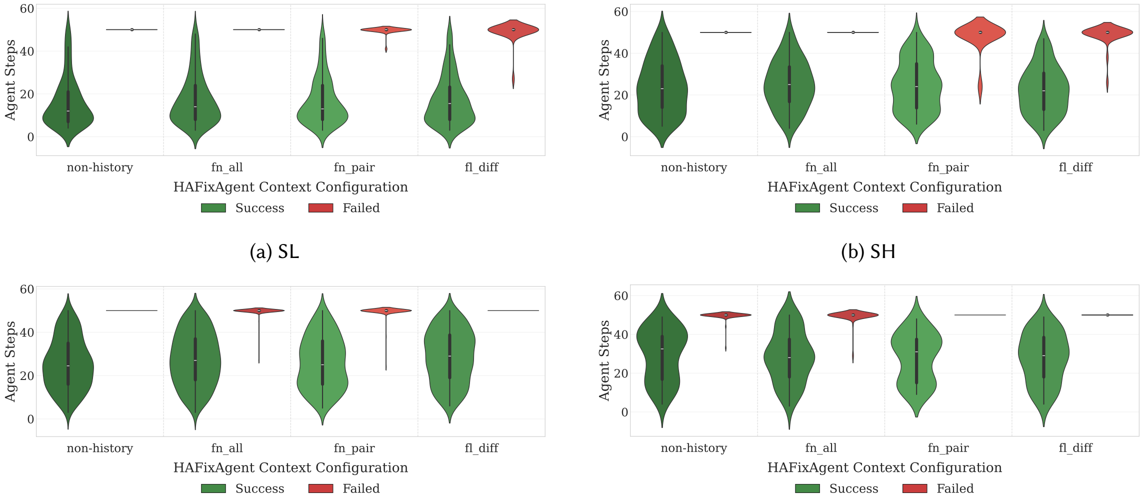
(c) SFMH

(d) MFMH

Fig. 3. Venn diagram comparing bug repair performance across 4 context configurations (*non-history*, *fn_all*, *fn_pair*, and *fl_diff*) over 4 bug categories on Defects4J.

6.2.2 Approach. We evaluate the inference cost (USD) and agent reasoning steps across all four configurations (*non-history*, *fn_all*, *fn_pair*, and *fl_diff*) over four bug categories (SL, SH, SFMH and MFMH). Inference cost is computed from input and output token usage converted to USD using official API pricing rates published for the DeepSeek-V3.2-Exp model. Efficiency is measured by the number of agent actions taken per run. A run terminates either upon a successful repair or by hitting one of the predefined limits (50 steps, $1 USD cost, or 1-hour timeout), as defined in Section 5.4. We report cost and step metrics separately for successful and failed attempts, allowing us to isolate and analyze the average resources required for a successful repair. Finally, we analyze the success rate versus cost across configurations to characterize category-specific trade-off frontiers.

6.2.3 Results. **Successful repairs converge well before the step cap.** Figure 4 reports agent step distributions by configuration and outcome. Across categories, the median number of steps for successful repairs falls far below 50, whereas for failed attempts it sits at the cap of 50 for all configurations. The median number of steps for successful repairs is 12-16 in SL, 22-25 in SH, 24-29 in SFMH, and 28-32 in MFMH, showing that harder categories require more steps but still finish well under the cap.



(a) SL

(b) SH

(c) SFMH

(d) MFMH

Median values (Success / Failed)

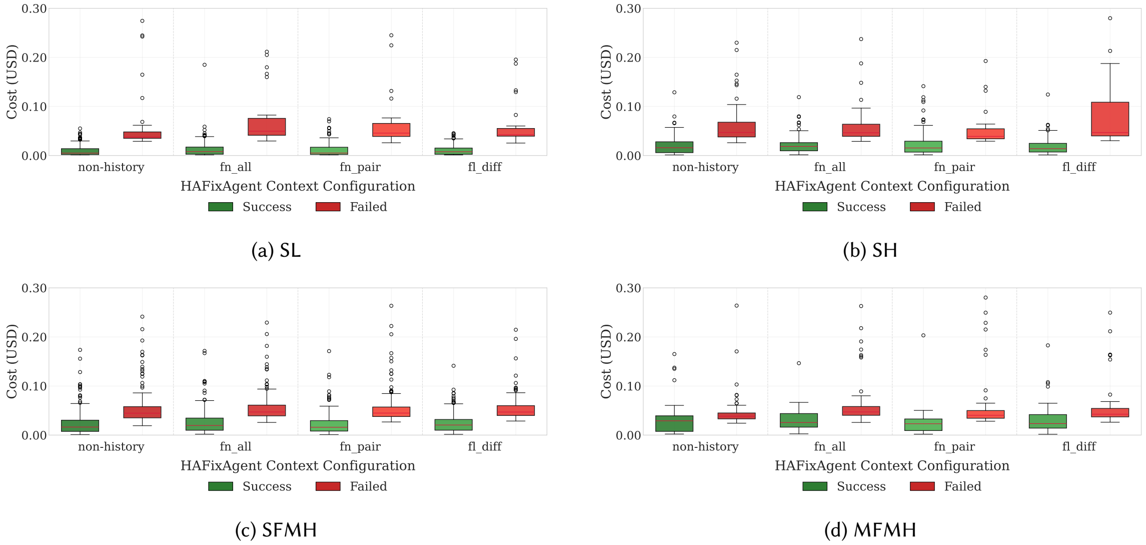
	SL	SH	SFMH	MFMH
non-history	12 / 50	23 / 50	24 / 50	32 / 50
fn_all	14 / 50	25 / 50	27 / 50	28 / 50
fn_pair	13 / 50	24 / 50	25 / 50	31 / 50
fl_diff	16 / 50	22 / 50	29 / 50	29 / 50

Fig. 4. Agent reasoning step distributions for successful and failed repairs across different HAFixAgent context configurations.

History does not significantly increase step counts. To rigorously assess whether historical heuristics significantly alter repair costs or the number of steps compared to the *non-history* configuration, we conducted Friedman tests followed by pairwise Wilcoxon signed-rank tests with Bonferroni correction ($\alpha = 0.0167$) on bugs successfully fixed by all four configurations (Table 5). Within each category, the success medians for *non-history*, *fn_all*, *fn_pair*, and *fl_diff* differ by only a few steps. The Friedman tests on this matched set detect no significant step differences in any category (all $p \geq 0.05$). Notably, in MFMH, the history configurations are as fast or faster than the *non-history* configuration (*non-history* 32 vs *fn_all* 28, *fn_pair* 31, *fl_diff* 29), and in SH, the *fl_diff* configuration converges the quickest (median 22). This confirms that adding historical context does not create a significant step overhead.

Agent cost is dominated by failure, not configuration. Figure 5 summarizes inference cost distributions by configuration and outcome. Across categories, failed attempts cost markedly more than successful ones, clustering near 0.03-0.05 USD median, while successes are cheaper: SL 0.005-0.008, SH 0.014-0.019, SFMH 0.016-0.021, and MFMH 0.023-0.029. This mirrors the step analysis, where failures hit the step cap and thus consume more tokens.

Adding historical context keeps the cost comparable in most configurations. For SL, the Friedman test is significant for cost, with the Wilcoxon post-hoc tests showing that *fn_all* ($p = 0.0022$) and *fl_diff* ($p = 0.001$) cost more than *non-history*, while *fn_pair* shows no significant difference ($p = 0.0444 > 0.0167$). For SH, SFMH, and MFMH, Friedman tests are not significant for cost (all $p \geq 0.05$). Notably, the median costs for successful repairs (shown in Figure 5) can be as low or lower with history in harder categories (for example, MFMH median success costs drop from



(a) SL

(b) SH

(c) SFMH

(d) MFMH

Median values (Success / Failed)

	SL	SH	SFMH	MFMH
non-history	0.005 / 0.038	0.016 / 0.047	0.016 / 0.045	0.029 / 0.038
fn_all	0.008 / 0.050	0.019 / 0.046	0.020 / 0.047	0.026 / 0.047
fn_pair	0.005 / 0.046	0.015 / 0.039	0.016 / 0.045	0.023 / 0.040
fl_diff	0.008 / 0.042	0.014 / 0.047	0.021 / 0.047	0.023 / 0.043

Fig. 5. Inference cost distributions for successful and failed repairs across different HAFixAgent context configurations.

Table 5. Statistical Comparison: History Heuristics vs. Non-history (Bonferroni $\alpha = 0.0167$). The last three columns report p-values for each history configuration compared against the *non-history* configuration.

Category	**Metric**	**N**	**Friedman p**	**Pairwise p-values**		
				fn_all	***fn_pair***	***fl_diff***
SL	Cost	118	< 0.001	**0.0022**	0.0044	**0.001**
	Steps	118	0.176		—	
SH	Cost	72	0.184		—	
	Steps	72	0.400			
SFMH	Cost	140	0.122		—	
	Steps	140	0.051			
MFMH	Cost	22	0.142		—	
	Steps	22	0.427			

N = number of bugs successfully fixed by all 4 configurations.
Bold values indicate $p < 0.0167$.
— indicates the Friedman test was not significant ($p \geq 0.05$); post-hoc tests were not performed.

0.029 to 0.026-0.023), but these reductions are not statistically significant on the matched set (i.e., the subset of bugs fixed by all four configurations).

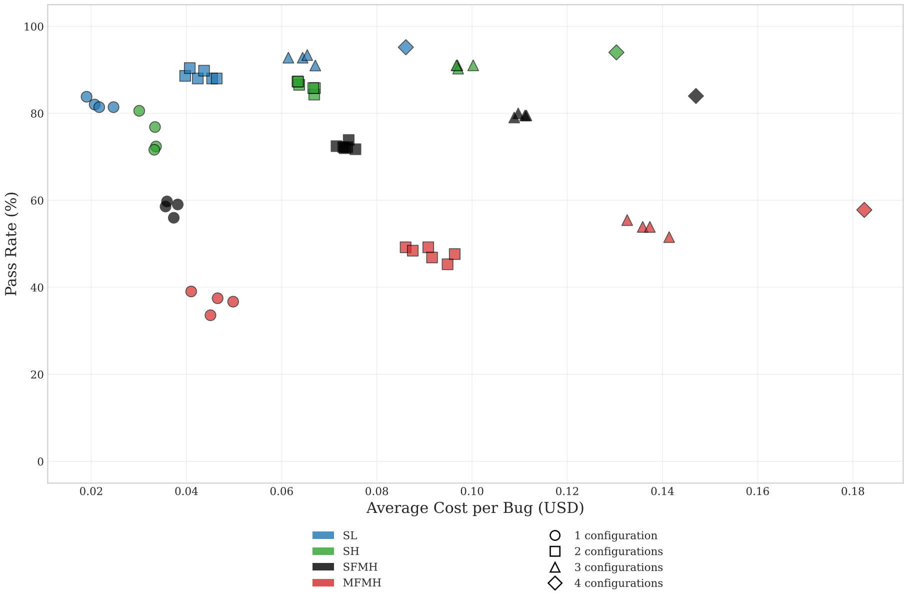


Fig. 6. Trade-off analysis of success rate and cost per bug across different combinations of configurations.

Combining history configurations increases success but also increases cost, exhibiting diminishing returns, and the most cost-effective choices are usually two or three configurations. Figure 6 plots success rate versus average cost per bug by category, with marker shape indicating how many configurations are combined. For Single-Line bugs, *fn_pair* +*fl_diff* reaches 90.4% at an average cost of 0.041 USD, close to the union of all four configurations, 95.2% at 0.086 USD. For Single-Hunk bugs, *fn_all* +*fn_pair* attains 87.3% at 0.063, while the four-configuration union gives 94.0% at 0.130. For Single-File-Multi-Hunk bugs, *fn_all* +*fl_diff* yields 73.9% at 0.074, compared to 84.0% at 0.147 with the four-configuration union. For Multi-File-Multi-Hunk bugs, *non-history* +*fn_pair* provides 49.2% at 0.086 before the cost rises to 57.8% at 0.182 with all four. Across categories, two or three history configurations give the best success per dollar. As these results show, adding the final configuration to create the four-configuration union raises the cost substantially for only small gains, demonstrating clear diminishing returns.

Summary for RQ2:

(1) Successful repairs finish well before the step cap, while failures commonly hit the cap.
(2) History-based APR agents are as efficient as *non-history* in terms of step count and cost (except for SL).
(3) Combining two or three history configurations delivers the best success, while using all configurations yields small extra gains at a significantly higher cost.

7 Discussion and Future Work

In this section, we provide a discussion of our results and introduce future work.

7.1 Performance Regressions

We observe 32 cases where a history configuration underperforms the *non-history* one or misses *non-history* fixes (Figure 3). For instance, in *MFMH*, the *non-history* configuration attains 50 while *fn_all*, *fn_pair*, and *fl_diff* reach 47, 43, and 48, respectively, even though each contributes 12 to 13 unique fixes that the *non-history* does not solve. This pattern aligns with LLM evidence that more context can hurt: models are easily distracted by irrelevant additions [59], and performance often declines when relevant facts sit in the middle or deeper parts of long prompts [37]. In our setting, extra historical snippets can dilute salient context and reduce Plausible@1 on some bugs that don't need historical context. Future work might filter and summarize history for direct relevance or add a guardrail that falls back to a leaner prompt when early steps show no progress.

7.2 Heuristics Complementarity

A key finding from RQ1 (Figure 3) is that the three historical heuristics are complementary, not redundant. Each historical heuristic (e.g., *fn_all*, *fn_pair*, *fl_diff*) repairs a distinct set of bugs that the others miss. This has important implications for why history is effective. It suggests that different bug types are susceptible to different forms of historical context:

- *fl_diff* (file-level diff), the strongest overall, likely excels at capturing fine-grained and localized edit patterns that are common in single-line and single-hunk bugs.
- *fn_pair* (historical function before and after) provides a richer semantic context, showing the agent how a function's logic was intended to operate, which may be crucial for more complex logic errors.
- *fn_all* (all functions from blame commit) captures co-evolutionary patterns (e.g., when function A changed, function B also changed), which is a vital clue for multi-hunk bugs where a fix requires coordinated edits.

This complementarity also explains the practical trade-off identified in RQ2 (Figure 6): combining two or three heuristics shows a higher pass rate and yields the most cost-effective performance, as it provides the agent with a diverse set of contexts.

7.3 Future Work

Richer historical heuristics beyond the blamed commit. HAFixAgent currently injects three compact heuristics drawn from the single commit that last touched the buggy lines. A broader temporal view may help: commits immediately before or after the blamed change, refactorings, and coevolution bursts can reveal the mechanism that produced the bug. Future work can build coevolution-aware retrieval over historical commit windows, summarize change intents such as API migrations or dependency upgrades, and learn when to expand beyond one commit. Such a broader historical context may improve robustness, particularly for bugs whose causes span multiple commits.

Enhanced context for blameless bugs. Building on our discussion in Section 4.1.3, the simple nearest-line fallback for the 28.9% of blameless bugs can be significantly improved. Instead of just finding the nearest line, a future agent could analyze the history of the parent class or module to infer project-specific patterns for adding new functionality. It could also search the repository for semantically similar add-only commits (e.g., how has this project added null-checks in other files?) to provide the agent with a template for the kind of code it needs to write.

Assessing history's impact on patch quality. A valuable future direction is to move beyond test-pass-based evaluation to investigate how historical context impacts the quality and maintainability of the generated patches. While multiple agent configurations might produce a plausible fix, it is an open question whether the history-aware patches are qualitatively superior. For instance, a patch generated with historical context might be more idiomatic, such as re-using a specific helper function or logging style from the historical diff, and more semantically aligned with the project's evolution. This could contrast with a *non-history* patch that, while functional, acts as a brute-force fix that simply passes the test suite. Future work could investigate this by designing automated tools to rate patch quality or by employing static analysis metrics to measure code complexity and semantic differences.

Extending to a multi-agent repair system. A promising, though more complex, frontier is to evolve HAFixAgent from a single-agent system into a collaborative multi-agent architecture [36, 41, 57]. Instead of one agent handling all tasks, we could design a system with specialized roles. For instance, a Coordinator agent could oversee the process, dispatching a specialized History agent to analyze repository patterns, blame commits, and issue reports to form a repair hypothesis. This hypothesis would then be passed to a Fixer agent focused on patch generation. Critically, a Reviewer agent could then critique the proposed patch, checking it not only against the test suite but also against the historical context for style, conventions, and potential side effects, which is similar to human code review. This separation of concerns could be particularly effective for the complex multi-file-multi-hunk (MFMH) bugs, where a single agent's reasoning chain and context window can be overwhelmed.

Applying historical context to other SE tasks beyond APR. The same history-based heuristics may benefit other SE code tasks such as code generation, code completion, code review, refactoring, and test generation. Repository history heuristics such as coevolution patterns, change intent, and dependency updates can guide model choices and keep outputs aligned with project conventions. Future work should evaluate these heuristics on tasks beyond repair using task-specific metrics (functional correctness, style consistency, temporal consistency, and safety), compare against retrieval baselines, and study generalization across languages and repositories.

8 Threats to Validity

In this section, we discuss the threats to validity of our study about HAFixAgent.

8.1 Internal Validity

Data leakage. DeepSeek-V3.2-Exp is available as open weights, but the pretraining corpus for this model is not disclosed to the best of our knowledge. As a result, we cannot verify if the specific projects or bugs in Defects4J are included in the pretraining data. However, our evaluation of HAFixAgent focuses on the relative effectiveness improvement of different historical heuristics, rather than their absolute performance. For instance, if HAFixAgent-*fl_diff* fixes a bug that the HAFixAgent-*non-history* does not fix, this improvement is attributed to the benefit of historical context, regardless of whether the LLM has seen the bug before. Our evaluation design naturally isolates and mitigates the possible influence of potential data leakage. To fully resolve this threat, we would need to retrain DeepSeek-V3.2-Exp from scratch, which would be infeasible for an academic project.

Fault localization. Our evaluation assumes perfect fault localization to isolate the evaluation of historical context on patch generation, a common setup in LLM-based APR studies [5, 14, 16, 51, 71, 75, 77]. Note that even given perfect fault localization, as most of the current work did, the LLM-based APR still struggle with complicated bugs (e.g., 39.1% Plausible for *non-history* configuration in MFMH bugs in our study). In practice, FL is noisy, especially for human-reported issues and multi-hunk defects, which can lower repair rates and lead to potentially wrong repair directions. For iterative agentic APR, updating FL during repair might help, but it does not eliminate the challenge. So we leverage perfect fault localization to isolate the evaluation of the historical context in HAFixAgent.

8.2 External Validity

Dataset and language generalizability. Our study uses Defects4J v3.0.1 with 854 Java bugs. Results may not transfer to other ecosystems where language, project size, test adequacy, and repository practices differ. The history profile we observe in Defects4J (for example, many bugs mapping to a single history commit) can change in other benchmarks, repositories with frequent code moves, or workflows that rewrite history. To mitigate this threat, we expose a dataset-agnostic interface behind the history extractor and provide dataset adapters that can be subclassed when porting to languages such as Python or C. We release code and scripts so other communities can rerun the pipeline on local datasets.

LLM and agent generalizability. HAFixAgent is instantiated with one agent loop, a narrow bash tool set, specific prompts, and a single LLM configuration with fixed step, cost, and time guards. Different agents, tool scopes, models, context windows, or decoding policies may change absolute numbers and the relative strength of the three history heuristics. API level determinism does not guarantee identical runs in practice, so small variation across executions and providers is expected. To reduce this threat, we use DeepSeek-V3.2-Exp as a recently released strong model at the time of writing. We also assume perfect fault localization to isolate the effect of history, and generalizing to noisy localization remains future work.

9 Related Work

In this section, we discuss related work about the traditional automated program repair, LLM and agent based automated program repair, and in-context learning for automated program repair.

9.1 Traditional Automated Program Repair

Traditional APR formulates patch generation as a search or constraint problem over program transformations guided by tests or specifications. Search-based systems evolve patches with genetic or heuristic search over mutation operators and validate candidates against the test suite, exemplified by GenProg and successors [33, 34, 67]. Semantics or constraint-based repair derives patches by solving synthesis constraints obtained from symbolic execution or program analysis, as in SemFix [52] and Angelix [48], and with targeted condition synthesis such as Nopol [74]. Template-based repair uses human-designed or mined fix schema that match buggy contexts and instantiate edits such as TBar [43] and AVATAR [42]. A complementary line reuses in-project code fragments as ingredients, such as ssFix [73] and CAPGEN [67]. These approaches established core pipelines for localization, candidate generation, and test-based validation, but their coverage is constrained by operator or pattern design and by search scalability.

Learning-based APR before LLMs frames repair as code-to-code translation or edit prediction trained on a large amount of code corpus. Early neural systems include DeepFix [19] for syntax errors and sequence-to-sequence repair, such as SequenceR [9], with later improvements from model ensembling and syntax guidance in CoCoNuT [45], CURE [27], and Recoder [82]. Template mining and ranking were also brought to production, for example, Facebook's Getafix [3] for static-analyzer warnings and the end-to-end SapFix [46] pipeline that proposed developer-reviewed patches. Together, traditional APR demonstrates effective repair across bug classes while revealing recurring limitations in computation cost, low accuracy and overfitting to tests. These limitations have motivated LLM and agent-based APR that learn fixes from data and plan tool-driven repair loops, which we introduce next.

9.2 LLM and Agent based Automated Program Repair

Large language models (LLMs) have significantly advanced Automated Program Repair (APR), moving from direct prompting to sophisticated, multi-step agentic systems. Early evidence shows code LLMs already outperform prior learning-based APR and benefit from fine-tuning and careful evaluation on standard benchmarks [26, 70]. This led to a variety of prompting and interaction strategies. For example, conversation-driven repair simulates a developer's debugging process by interleaving patch generation with test feedback, achieving a strong balance of cost and accuracy [72]. Other approaches focus on improving the model's reasoning, such as self-directed repair, which uses a chain-of-thought process to gather knowledge before fixing [78]. The plastic surgery line revisits ingredient reuse by aligning LLMs with project-specific code [69]. Beyond pure prompting, hybrid and template-guided methods constrain or scaffold the patch space with analysis or repair templates to improve plausibility and correctness [25, 38].

Agent-based APR systems represent a further step, shifting from single-shot patch generation to autonomous, tool-using workflows. Pioneering general-purpose software agents, like SWE-agent [75] and OpenHands [65], established a powerful paradigm by equipping LLMs with tools to interact with a repository, such as editors, shell commands, and test runners. Building on this, repair-focused agents have integrated more specialized tools. For instance, some employ search and fault localization to narrow down the buggy code [5, 81], while others explore multi-agent collaboration to divide the debugging task [36], with empirical studies beginning to map this rapidly evolving design space [49]. A recent trend also focuses on improving agent performance on complex, repository-level tasks by incorporating memory and experience, allowing agents to learn from prior repair trajectories [7, 50]. These lines generally rely on local code and runtime outputs, with limited study of commit history as a first-class context inside the loop, which is the focus of our work.

9.3 In-context Learning for Automated Program Repair

A core question for LLM-based APR is which context to provide and how to structure it. Prior work shows that adding bug-related facts such as error messages, stack traces, and failing tests improves repair, and that the choice of facts matters [55, 72]. Studies on local context indicate that models are sensitive to how much and which code surrounds the edit region [56]. Other work leverages natural language bug reports or transforms descriptions into edits, and explores execution traces to guide patching [15, 20]. Repository-level evaluations report that larger, realistic tasks remain challenging and require broader context curation [8].

Beyond local signals, historical context is receiving attention. HAFix [60] demonstrates that blame-derived commit data is an effective, lightweight historical signal for fixing more bugs, motivating history-aware agent designs at Defects4J scale. Layered knowledge injection systematically adds project and repository knowledge to prompts and improves bug fixing on broader-context scenarios [14]. Built on top of HAFix, HAFixAgent advances this direction by bringing blame-driven history into the agentic workflow and quantifying its effect on single-hunk and multi-hunk bugs.

10 Conclusion

This paper presented HAFixAgent, a history-aware agentic approach that injects blame-derived repository context into an observe-act-verify loop to address the more complex single- and multi-hunk bug categories. A preliminary study shows how blame commit history is not only widely available (71.1% blameable), but also is surprisingly concentrated with 70.7% of Defects4J bugs having exactly one unique blame commit across their buggy line.

When applying HaFixAgent on all 854 Defects4J bugs, history-aware settings were able to add 194 history-only fixes that the *non-history* configuration never achieves. When compared to SOTA baselines, HAFixAgent outperforms RepairAgent by an average of 212.3% and BIRCH feedback by 29.9%. Among the three historical contexts, *fl_diff* is the strongest overall and *fn_pair* leads on single-hunk bugs. Successful runs finish well before the 50-step cap, and the median token cost does not increase; in the hardest multi-file-multi-hunk category, costs are lower than the *non-history* configuration. Lastly, combining two or three heuristics captures most gains, while leveraging all heuristics adds little benefit for a significantly higher cost. These findings give a simple recipe for practice: (1) ground the agent loop in version control history, (2) prefer diff-based context, (3) add one complementary heuristic when needed, and (4) avoid collecting excessive context.

This study has natural limitations. We assume perfect fault localization (similar to related work), focus on one benchmark and language, and use a single strong agent model and loop. Future work will relax these assumptions by evaluating additional datasets and languages, testing imperfect localization, learning to select or combine history heuristics at runtime, and integrating richer project artifacts. We expect progressive use of historical context to remain a robust and lightweight direction for advancing agent-based program repair.

References

[1] Bram Adams, Zhen Ming Jiang, and Ahmed E. Hassan. 2010. Identifying crosscutting concerns using historical code changes. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE 2010, Cape Town, South Africa, 1-8 May 2010*, Jeff Kramer, Judith Bishop, Premkumar T. Devanbu, and Sebastián Uchitel (Eds.). ACM, 305–314. doi:10.1145/1806799.1806846

[2] Anthropic. 2025. Claude Code Overview. https://docs.anthropic.com/en/docs/claude-code/overview Accessed: 2025-10-28.

[3] Johannes Bader, Andrew Scott, Michael Pradel, and Satish Chandra. 2019. Getafix: learning to fix bugs automatically. *Proc. ACM Program. Lang.* 3, OOPSLA (2019), 159:1–159:27. doi:10.1145/3360585

[4] CO BOULDER. 2013. *University of Cambridge Study: Failure to Adopt Reverse Debugging Costs Global Economy $41 Billion Annually.* https://finance.yahoo.com/news/university-cambridge-study-failure-adopt-110000587.html Accessed: 2025-10-26.

[5] Islem Bouzenia, Premkumar T. Devanbu, and Michael Pradel. 2025. RepairAgent: An Autonomous, LLM-Based Agent for Program Repair. In *47th IEEE/ACM International Conference on Software Engineering, ICSE 2025, Ottawa, ON, Canada, April 26 - May 6, 2025.* IEEE, 2188–2200. doi:10.1109/ICSE55347.2025.00157

[6] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating Large Language Models Trained on Code. *CoRR* abs/2107.03374 (2021). arXiv:2107.03374 https://arxiv.org/abs/2107.03374

[7] Silin Chen, Shaoxin Lin, Xiaodong Gu, Yuling Shi, Heng Lian, Longfei Yun, Dong Chen, Weiguo Sun, Lin Cao, and Qianxiang Wang. 2025. SWE-Exp: Experience-Driven Software Issue Resolution. *CoRR* abs/2507.23361 (2025). arXiv:2507.23361 doi:10.48550/ARXIV.2507.23361

[8] Yuxiao Chen, Jingzheng Wu, Xiang Ling, Changjiang Li, Zhiqing Rui, Tianyue Luo, and Yanjun Wu. 2024. When Large Language Models Confront Repository-Level Automatic Program Repair: How Well They Done?. In *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings, ICSE Companion 2024, Lisbon, Portugal, April 14-20, 2024.* ACM, 459–471. doi:10.1145/3639478.3647633

[9] Zimin Chen, Steve Kommrusch, Michele Tufano, Louis-Noël Pouchet, Denys Poshyvanyk, and Martin Monperrus. 2021. SequenceR: Sequence-to-Sequence Learning for End-to-End Program Repair. *IEEE Trans. Software Eng.* 47, 9 (2021), 1943–1959. doi:10.1109/TSE.2019.2940179

[10] Inc. DeepSeek. 2025. *The Temperature Parameter.* https://api-docs.deepseek.com/quick_start/parameter_settings DeepSeek API Docs, Quick Start. Accessed 2025-10-20.

[11] DeepSeek-AI. 2025. *DeepSeek-V3.2-Exp Technical Note.* Technical Report. DeepSeek, Inc. https://github.com/deepseek-ai/DeepSeek-V3.2-Exp/blob/main/DeepSeek_V3_2.pdf Accessed 2025-10-20.

[12] DeepSeek-AI, Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, Damai Dai, Daya Guo, Dejian Yang, Deli Chen, Dongjie Ji, Erhang Li, Fangyun Lin, Fucong Dai, Fuli Luo, Guangbo Hao, Guanting Chen, Guowei Li, H. Zhang, Han Bao, Hanwei Xu, Haocheng Wang, Haowei Zhang, Honghui Ding, Huajian Xin, Huazuo Gao, Hui Li, Hui Qu, J. L. Cai, Jian Liang, Jianzhong Guo, Jiaqi Ni, Jiashi Li, Jiawei Wang, Jin Chen, Jingchang Chen, Jingyang Yuan, Junjie Qiu, Junlong Li, Junxiao Song, Kai Dong, Kai Hu, Kaige Gao, Kang Guan, Kexin Huang, Kuai Yu, Lean Wang, Lecong Zhang, Lei Xu, Leyi Xia, Liang Zhao, Litong Wang, Liyue Zhang, Meng Li, Miaojun Wang, Mingchuan Zhang, Minghua Zhang, Minghui Tang, Mingming Li, Ning Tian, Panpan Huang, Peiyi Wang, Peng Zhang, Qiancheng Wang, Qihao Zhu, Qinyu Chen, Qiushi Du, R. J. Chen, R. L. Jin, Ruiqi Ge, Ruisong Zhang, Ruizhe Pan, Runji Wang, Runxin Xu, Ruoyu Zhang, Ruyi Chen, S. S. Li, Shanghao Lu, Shangyan Zhou, Shanhuang Chen, Shaoqing Wu, Shengfeng Ye, Shirong Ma, Shiyu Wang, Shuang Zhou, Shuiping Yu, Shunfeng Zhou, Shuting Pan, T. Wang, Tao Yun, Tian Pei, Tianyu Sun, W. L. Xiao, and Wangding Zeng. 2024. DeepSeek-V3 Technical Report. *CoRR* abs/2412.19437 (2024). arXiv:2412.19437 doi:10.48550/ARXIV.2412.19437

[13] Olive Jean Dunn. 1961. Multiple comparisons among means. *Journal of the American statistical association* 56, 293 (1961), 52–64.

[14] Ramtin Ehsani, Esteban Parra, Sonia Haiduc, and Preetha Chatterjee. 2025. Bug Fixing with Broader Context: Enhancing LLM-Based Program Repair via Layered Knowledge Injection. *CoRR* abs/2506.24015 (2025). arXiv:2506.24015 doi:10.48550/ARXIV.2506.24015

[15] Sarah Fakhoury, Saikat Chakraborty, Madanlal Musuvathi, and Shuvendu K. Lahiri. 2024. NL2Fix: Generating Functionally Correct Code Edits from Bug Descriptions. In *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings, ICSE Companion 2024, Lisbon, Portugal, April 14-20, 2024*. ACM, 410–411. doi:10.1145/3639478.3643526

[16] Zhiyu Fan, Xiang Gao, Martin Mirchev, Abhik Roychoudhury, and Shin Hwei Tan. 2023. Automated Repair of Programs from Large Language Models. In *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*. IEEE, 1469–1481. doi:10.1109/ICSE48619.2023.00128

[17] Zhiyu Fan, Kirill Vasilevski, Dayi Lin, Boyuan Chen, Yihao Chen, Zhiqing Zhong, Jie M. Zhang, Pinjia He, and Ahmed E. Hassan. 2025. SWE-Effi: Re-Evaluating Software AI Agent System Effectiveness Under Resource Constraints. *CoRR* abs/2509.09853 (2025). arXiv:2509.09853 doi:10.48550/ARXIV.2509.09853

[18] Milton Friedman. 1937. The use of ranks to avoid the assumption of normality implicit in the analysis of variance. *Journal of the american statistical association* 32, 200 (1937), 675–701.

[19] Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish K. Shevade. 2017. DeepFix: Fixing Common C Language Errors by Deep Learning. In *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence, February 4-9, 2017, San Francisco, California, USA*, Satinder Singh and Shaul Markovitch (Eds.). AAAI Press, 1345–1351. doi:10.1609/AAAI.V31I1.10742

[20] Mirazul Haque, Petr Babkin, Farima Farmahinifarahani, and Manuela Veloso. 2025. Towards Effectively Leveraging Execution Traces for Program Repair with Code LLMs. *CoRR* abs/2505.04441 (2025). arXiv:2505.04441 doi:10.48550/ARXIV.2505.04441

[21] Ahmed E. Hassan. 2006. Mining Software Repositories to Assist Developers and Support Managers. In *22nd IEEE International Conference on Software Maintenance (ICSM 2006), 24-27 September 2006, Philadelphia, Pennsylvania, USA*. IEEE Computer Society, 339–342. doi:10.1109/ICSM.2006.38

[22] Ahmed E. Hassan, Hao Li, Dayi Lin, Bram Adams, Tse-Hsun Chen, Yutaro Kashiwa, and Dong Qiu. 2025. Agentic Software Engineering: Foundational Pillars and a Research Roadmap. *CoRR* abs/2509.06216 (2025). arXiv:2509.06216 doi:10.48550/ARXIV.2509.06216

[23] Ahmed E. Hassan, Dayi Lin, Gopi Krishnan Rajbahadur, Keheliya Gallaba, Filipe Roseiro Côgo, Boyuan Chen, Haoxiang Zhang, Kishanthan Thangarajah, Gustavo Ansaldi Oliva, Jiahuei (Justina) Lin, Wali Mohammad Abdullah, and Zhen Ming (Jack) Jiang. 2024. Rethinking Software Engineering in the Era of Foundation Models: A Curated Catalogue of Challenges in the Development of Trustworthy FMware. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering, FSE 2024, Porto de Galinhas, Brazil, July 15-19, 2024*, Marcelo d'Amorim (Ed.). ACM, 294–305. doi:10.1145/3663529.3663849

[24] Soneya Binta Hossain, Nan Jiang, Qiang Zhou, Xiaopeng Li, Wen-Hao Chiang, Yingjun Lyu, Hoan Anh Nguyen, and Omer Tripp. 2024. A Deep Dive into Large Language Models for Automated Bug Localization and Repair. *Proc. ACM Softw. Eng.* 1, FSE (2024), 1471–1493. doi:10.1145/3660773

[25] Kai Huang, Jian Zhang, Xiangxin Meng, and Yang Liu. 2025. Template-Guided Program Repair in the Era of Large Language Models. In *47th IEEE/ACM International Conference on Software Engineering, ICSE 2025, Ottawa, ON, Canada, April 26 - May 6, 2025*. IEEE, 1895–1907. doi:10.1109/ICSE55347.2025.00030

[26] Nan Jiang, Kevin Liu, Thibaud Lutellier, and Lin Tan. 2023. Impact of Code Language Models on Automated Program Repair. In *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*. IEEE, 1430–1442. doi:10.1109/ICSE48619.2023.00125

[27] Nan Jiang, Thibaud Lutellier, and Lin Tan. 2021. CURE: Code-Aware Neural Machine Translation for Automatic Program Repair. In *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021*. IEEE, 1161–1173. doi:10.1109/ICSE43902.2021.00107

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

[29] René Just, Darioush Jalali, and Michael D. Ernst. 2014. Defects4J: a database of existing faults to enable controlled testing studies for Java programs. In *International Symposium on Software Testing and Analysis, ISSTA '14, San Jose, CA, USA - July 21 - 26, 2014*, Corina S. Pasareanu and Darko Marinov (Eds.). ACM, 437–440. doi:10.1145/2610384.2628055

[30] Yasutaka Kamei, Emad Shihab, Bram Adams, Ahmed E. Hassan, Audris Mockus, Anand Sinha, and Naoyasu Ubayashi. 2013. A Large-Scale Empirical Study of Just-in-Time Quality Assurance. *IEEE Trans. Software Eng.* 39, 6 (2013), 757–773. doi:10.1109/TSE.2012.70

[31] Sunghun Kim, Thomas Zimmermann, Kai Pan, and E. James Whitehead Jr. 2006. Automatic Identification of Bug-Introducing Changes. In *21st IEEE/ACM International Conference on Automated Software Engineering (ASE 2006), 18-22 September 2006, Tokyo, Japan*. IEEE Computer Society, 81–90. doi:10.1109/ASE.2006.23

[32] Xuan-Bach Dinh Le, Duc-Hiep Chu, David Lo, Claire Le Goues, and Willem Visser. 2017. S3: syntax- and semantic-guided repair synthesis via programming by examples. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*, Eric Bodden, Wilhelm Schäfer, Arie van Deursen, and Andrea Zisman (Eds.). ACM, 593–604. doi:10.1145/3106237.3106309

[33] Xuan-Bach Dinh Le, David Lo, and Claire Le Goues. 2016. History Driven Program Repair. In *IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering, SANER 2016, Suita, Osaka, Japan, March 14-18, 2016 - Volume 1*. IEEE Computer Society, 213–224. doi:10.1109/SANER.2016.76

[34] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. 2012. GenProg: A Generic Method for Automatic Software Repair. *IEEE Trans. Software Eng.* 38, 1 (2012), 54–72. doi:10.1109/TSE.2011.104

[35] Claire Le Goues, Michael Pradel, and Abhik Roychoudhury. 2019. Automated program repair. *Commun. ACM* 62, 12 (2019), 56–65. doi:10.1145/3318162

[36] Cheryl Lee, Chunqiu Steven Xia, Jen-tse Huang, Zhouruixin Zhu, Lingming Zhang, and Michael R. Lyu. 2024. A Unified Debugging Approach via LLM-Based Multi-Agent Synergy. *CoRR* abs/2404.17153 (2024). arXiv:2404.17153 doi:10.48550/ARXIV.2404.17153

[37] Mosh Levy, Alon Jacoby, and Yoav Goldberg. 2024. Same Task, More Tokens: the Impact of Input Length on the Reasoning Performance of Large Language Models. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2024, Bangkok, Thailand, August 11-16, 2024*, Lun-Wei Ku, Andre Martins, and Vivek Srikumar (Eds.). Association for Computational Linguistics, 15339–15353. doi:10.18653/V1/2024.ACL-LONG.818

[38] Fengjie Li, Jiajun Jiang, Jiajun Sun, and Hongyu Zhang. 2025. Hybrid Automated Program Repair by Combining Large Language Models and Program Analysis. *ACM Trans. Softw. Eng. Methodol.* 34, 7 (2025), 202:1–202:28. doi:10.1145/3715004

[39] Hao Li, Haoxiang Zhang, and Ahmed E. Hassan. 2025. The Rise of AI Teammates in Software Engineering (SE) 3.0: How Autonomous Coding Agents Are Reshaping Software Engineering. *CoRR* abs/2507.15003 (2025). arXiv:2507.15003 doi:10.48550/ARXIV.2507.15003

[40] Yi Li, Shaohua Wang, and Tien N. Nguyen. 2020. DLFix: context-based code transformation learning for automated program repair. In *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*, Gregg Rothermel and Doo-Hwan Bae (Eds.). ACM, 602–614. doi:10.1145/3377811.3380345

[41] Feng Lin, Dong Jae Kim, and Tse-Hsun Chen. 2025. SOEN-101: Code Generation by Emulating Software Process Models Using Large Language Model Agents. In *47th IEEE/ACM International Conference on Software Engineering, ICSE 2025, Ottawa, ON, Canada, April 26 - May 6, 2025*. IEEE, 1527–1539. doi:10.1109/ICSE55347.2025.00140

[42] Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F. Bissyandé. 2019. AVATAR: Fixing Semantic Bugs with Fix Patterns of Static Analysis Violations. In *26th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2019, Hangzhou, China, February 24-27, 2019*, Xinyu Wang, David Lo, and Emad Shihab (Eds.). IEEE, 456–467. doi:10.1109/SANER.2019.8667970

[43] Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F. Bissyandé. 2019. TBar: revisiting template-based automated program repair. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2019, Beijing, China, July 15-19, 2019*, Dongmei Zhang and Anders Møller (Eds.). ACM, 31–42. doi:10.1145/3293882.3330577

[44] Fan Long and Martin C. Rinard. 2015. Staged program repair with condition synthesis. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*, Elisabetta Di Nitto, Mark Harman, and Patrick Heymans (Eds.). ACM, 166–178. doi:10.1145/2786805.2786811

[45] Thibaud Lutellier, Hung Viet Pham, Lawrence Pang, Yitong Li, Moshi Wei, and Lin Tan. 2020. CoCoNuT: combining context-aware neural translation models using ensemble for program repair. In *ISSTA '20: 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, USA, July 18-22, 2020*, Sarfraz Khurshid and Corina S. Pasareanu (Eds.). ACM, 101–114. doi:10.1145/3395363.3397369

[46] Alexandru Marginean, Johannes Bader, Satish Chandra, Mark Harman, Yue Jia, Ke Mao, Alexander Mols, and Andrew Scott. 2019. SapFix: automated end-to-end repair at scale. In *Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Practice, ICSE (SEIP) 2019, Montreal, QC, Canada, May 25-31, 2019*, Helen Sharp and Mike Whalen (Eds.). IEEE / ACM, 269–278. doi:10.1109/ICSE-SEIP.2019.00039

[47] Matias Martinez and Martin Monperrus. 2016. ASTOR: a program repair library for Java (demo). In *Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, Saarbrücken, Germany, July 18-20, 2016*, Andreas Zeller and Abhik Roychoudhury (Eds.). ACM, 441–444. doi:10.1145/2931037.2948705

[48] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2016. Angelix: scalable multiline program patch synthesis via symbolic analysis. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*, Laura K. Dillon, Willem Visser, and Laurie A. Williams (Eds.). ACM, 691–701. doi:10.1145/2884781.2884807

[49] Xiangxin Meng, Zexiong Ma, Pengfei Gao, and Chao Peng. 2024. An Empirical Study on LLM-based Agents for Automated Bug Fixing. *CoRR* abs/2411.10213 (2024). arXiv:2411.10213 doi:10.48550/ARXIV.2411.10213

[50] Fangwen Mu, Junjie Wang, Lin Shi, Song Wang, Shoubin Li, and Qing Wang. 2025. EXPEREPAIR: Dual-Memory Enhanced LLM-based Repository-Level Program Repair. *CoRR* abs/2506.10484 (2025). arXiv:2506.10484 doi:10.48550/ARXIV.2506.10484

[51] Noor Nashid, Daniel Ding, Keheliya Gallaba, Ahmed E. Hassan, and Ali Mesbah. 2025. Characterizing Multi-Hunk Patches: Divergence, Proximity, and LLM Repair Challenges. *CoRR* abs/2506.04418 (2025). arXiv:2506.04418 doi:10.48550/ARXIV.2506.04418

[52] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. 2013. SemFix: program repair via semantic analysis. In *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*, David Notkin, Betty H. C. Cheng, and Klaus Pohl (Eds.). IEEE Computer Society, 772–781. doi:10.1109/ICSE.2013.6606623

[53] OpenAI. 2025. GPT-5 System Card. https://cdn.openai.com/gpt-5-system-card.pdf. Accessed: 2025-10-30.

[54] OpenAI. 2025. Introducing Codex. https://openai.com/index/introducing-codex/ Accessed: 2025-10-28.

[55] Nikhil Parasaram, Huijie Yan, Boyu Yang, Zineb Flahy, Abriele Qudsi, Damian Ziaber, Earl T. Barr, and Sergey Mechtaev. 2025. The Fact Selection Problem in LLM-Based Program Repair. In *47th IEEE/ACM International Conference on Software Engineering, ICSE 2025, Ottawa, ON, Canada, April 26 - May 6, 2025*. IEEE, 2574–2586. doi:10.1109/ICSE55347.2025.00162

[56] Julian Aron Prenner and Romain Robbes. 2024. Out of Context: How important is Local Context in Neural Program Repair?. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, April 14-20, 2024*. ACM, 83:1–83:13. doi:10.1145/3597503.3639086

[57] Md Nakhla Rafi, Dong Jae Kim, Tse-Hsun Chen, and Shaowei Wang. 2024. A Multi-Agent Approach to Fault Localization via Graph-Based Retrieval and Reflexion. *arXiv preprint arXiv:2409.13642* (2024).

[58] Abhik Roychoudhury. 2025. Agentic AI for Software: thoughts from Software Engineering community. *CoRR* abs/2508.17343 (2025). arXiv:2508.17343 doi:10.48550/ARXIV.2508.17343

[59] Freda Shi, Xinyun Chen, Kanishka Misra, Nathan Scales, David Dohan, Ed H. Chi, Nathanael Schärli, and Denny Zhou. 2023. Large Language Models Can Be Easily Distracted by Irrelevant Context. In *International Conference on Machine Learning, ICML 2023, 23-29 July 2023, Honolulu, Hawaii, USA (Proceedings of Machine Learning Research, Vol. 202)*, Andreas Krause, Emma Brunskill, Kyunghyun Cho, Barbara Engelhardt, Sivan Sabato, and Jonathan Scarlett (Eds.). PMLR, 31210–31227. https://proceedings.mlr.press/v202/shi23a.html

[60] Yu Shi, Abdul Ali Bangash, Emad Fallahzadeh, Bram Adams, and Ahmed E. Hassan. 2025. HAFix: History-Augmented Large Language Models for Bug Fixing. *CoRR* abs/2501.09135 (2025). arXiv:2501.09135 doi:10.48550/ARXIV.2501.09135

[61] André Silva and Martin Monperrus. 2025. RepairBench: Leaderboard of Frontier Models for Program Repair. In *IEEE/ACM International Workshop on Large Language Models for Code, LLM4Code@ICSE 2025, Ottawa, ON, Canada, May 3, 2025*. IEEE, 9–16. doi:10.1109/LLM4CODE66737.2025.00006

[62] Mifta Sintaha, Noor Nashid, and Ali Mesbah. 2023. Katana: Dual Slicing Based Context for Learning Bug Fixes. *ACM Trans. Softw. Eng. Methodol.* 32, 4 (2023), 100:1–100:27. doi:10.1145/3579640

[63] Jacek Sliwerski, Thomas Zimmermann, and Andreas Zeller. 2005. When do changes induce fixes?. In *Proceedings of the 2005 International Workshop on Mining Software Repositories, MSR 2005, Saint Louis, Missouri, USA, May 17, 2005*. ACM. doi:10.1145/1083142.1083147

[64] Lei Wang, Chen Ma, Xueyang Feng, Zeyu Zhang, Hao Yang, Jingsen Zhang, Zhiyuan Chen, Jiakai Tang, Xu Chen, Yankai Lin, Wayne Xin Zhao, Zhewei Wei, and Jirong Wen. 2024. A survey on large language model based autonomous agents. *Frontiers Comput. Sci.* 18, 6 (2024), 186345. doi:10.1007/S11704-024-40231-1

[65] 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, and et al. 2025. OpenHands: An Open Platform for AI Software Developers as Generalist Agents. In *The Thirteenth International Conference on Learning Representations, ICLR 2025, Singapore, April 24-28, 2025*. OpenReview.net. https://openreview.net/forum?id=OJd3ayDDoF

[66] Cathrin Weiß, Rahul Premraj, Thomas Zimmermann, and Andreas Zeller. 2007. How Long Will It Take to Fix This Bug?. In *Fourth International Workshop on Mining Software Repositories, MSR 2007 (ICSE Workshop), Minneapolis, MN, USA, May 19-20, 2007, Proceedings*. IEEE Computer Society, 1. doi:10.1109/MSR.2007.13

[67] Ming Wen, Junjie Chen, Rongxin Wu, Dan Hao, and Shing-Chi Cheung. 2018. Context-aware patch generation for better automated program repair. In *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, Michel Chaudron, Ivica Crnkovic, Marsha Chechik, and Mark Harman (Eds.). ACM, 1–11. doi:10.1145/3180155.3180233

[68] Frank Wilcoxon. 1992. Individual comparisons by ranking methods. In *Breakthroughs in statistics: Methodology and distribution*. Springer, 196–202.

[69] Chunqiu Steven Xia, Yifeng Ding, and Lingming Zhang. 2023. The Plastic Surgery Hypothesis in the Era of Large Language Models. In *38th IEEE/ACM International Conference on Automated Software Engineering, ASE 2023, Luxembourg, September 11-15, 2023*. IEEE, 522–534. doi:10.1109/ASE56229.2023.00047

[70] Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. 2023. Automated Program Repair in the Era of Large Pre-trained Language Models. In *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*. IEEE, 1482–1494. doi:10.1109/ICSE48619.2023.00129

[71] Chunqiu Steven Xia and Lingming Zhang. 2022. Less training, more repairing please: revisiting automated program repair via zero-shot learning. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2022, Singapore, Singapore, November 14-18, 2022*, Abhik Roychoudhury, Cristian Cadar, and Miryung Kim (Eds.). ACM, 959–971. doi:10.1145/3540250.3549101

[72] Chunqiu Steven Xia and Lingming Zhang. 2024. 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, ISSTA 2024, Vienna, Austria, September 16-20, 2024*, Maria Christakis and Michael Pradel (Eds.). ACM, 819–831. doi:10.1145/3650212.3680323

[73] Qi Xin and Steven P. Reiss. 2017. Leveraging syntax-related code for automated program repair. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017*, Grigore Rosu, Massimiliano Di Penta, and

Tien N. Nguyen (Eds.). IEEE Computer Society, 660–670. doi:10.1109/ASE.2017.8115676

[74] Jifeng Xuan, Matias Martinez, Favio Demarco, Maxime Clement, Sebastian R. Lamelas Marcote, Thomas Durieux, Daniel Le Berre, and Martin Monperrus. 2017. Nopol: Automatic Repair of Conditional Statement Bugs in Java Programs. *IEEE Trans. Software Eng.* 43, 1 (2017), 34–55. doi:10.1109/TSE.2016.2560811

[75] John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. 2024. SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering. In *Advances in Neural Information Processing Systems 38: Annual Conference on Neural Information Processing Systems 2024, NeurIPS 2024, Vancouver, BC, Canada, December 10 - 15, 2024*, Amir Globersons, Lester Mackey, Danielle Belgrave, Angela Fan, Ulrich Paquet, Jakub M. Tomczak, and Cheng Zhang (Eds.). http://papers.nips.cc/paper_files/paper/2024/hash/5a7c947568c1b1328ccc5230172e1e7c-Abstract-Conference.html

[76] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R. Narasimhan, and Yuan Cao. 2023. ReAct: Synergizing Reasoning and Acting in Language Models. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net. https://openreview.net/forum?id=WE_vluYUL-X

[77] He Ye and Martin Monperrus. 2024. ITER: Iterative Neural Repair for Multi-Location Patches. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, April 14-20, 2024*. ACM, 10:1–10:13. doi:10.1145/3597503.3623337

[78] Xin Yin, Chao Ni, Shaohua Wang, Zhenhao Li, Limin Zeng, and Xiaohu Yang. 2024. ThinkRepair: Self-Directed Automated Program Repair. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2024, Vienna, Austria, September 16-20, 2024*, Maria Christakis and Michael Pradel (Eds.). ACM, 1274–1286. doi:10.1145/3650212.3680359

[79] Quanjun Zhang, Chunrong Fang, Yuxiang Ma, Weisong Sun, and Zhenyu Chen. 2024. A Survey of Learning-based Automated Program Repair. *ACM Trans. Softw. Eng. Methodol.* 33, 2 (2024), 55:1–55:69. doi:10.1145/3631974

[80] Quanjun Zhang, Chunrong Fang, Yang Xie, Yuxiang Ma, Weisong Sun, Yun Yang, and Zhenyu Chen. 2024. A Systematic Literature Review on Large Language Models for Automated Program Repair. *CoRR* abs/2405.01466 (2024). arXiv:2405.01466 doi:10.48550/ARXIV.2405.01466

[81] Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. 2024. AutoCodeRover: Autonomous Program Improvement. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2024, Vienna, Austria, September 16-20, 2024*, Maria Christakis and Michael Pradel (Eds.). ACM, 1592–1604. doi:10.1145/3650212.3680384

[82] Qihao Zhu, Zeyu Sun, Yuan-an Xiao, Wenjie Zhang, Kang Yuan, Yingfei Xiong, and Lu Zhang. 2021. A syntax-guided edit decoder for neural program repair. In *ESEC/FSE '21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens, Greece, August 23-28, 2021*, Diomidis Spinellis, Georgios Gousios, Marsha Chechik, and Massimiliano Di Penta (Eds.). ACM, 341–353. doi:10.1145/3468264.3468544

[83] Qihao Zhu, Zeyu Sun, Wenjie Zhang, Yingfei Xiong, and Lu Zhang. 2023. Tare: Type-Aware Neural Program Repair. In *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*. IEEE, 1443–1455. doi:10.1109/ICSE48619.2023.00126

A HAFixAgent Prompt

System Prompt of HAFixAgent. The values inside placeholder will be rendered in runtime.

~~~
You are HAFixAgent, an expert Java debugging assistant specializing in Defects4J bug repair.

Your response must contain exactly ONE bash code block with ONE command (or commands connected with && or ||).

```bash
your_command_here
```

# Environment & Tools
You operate in a Linux environment with the buggy project checked out at `{{ repo_path }}`. You have full bash access
       for:

## File Operations
- **Overview**: `grep -n "class \|interface \|public.*(" file.java` (show classes and methods)
- **Targeted reading**: Read around fault locations with `sed -n` for specific line ranges
- **Progressive context**: Start with ±10 lines, expand to ±15, ±20, ±25 as needed
~~~

```
- **Edit**: Precise editing with `sed -i` (simple changes) or `head`/`tail` reconstruction (complex multi-line changes)
- **Search**: Find code patterns with `grep`, locate files with `find`

## Defects4J Commands
- **Compile**: `defects4j compile` - Initial compilation to verify setup
- **Test**: `defects4j test -r` - Compile and run relevant/failing tests

# Bug Fixing Methodology

## 1. Understand the Bug
- Read the bug description and fault locations carefully
- Examine failing test cases to understand expected vs actual behavior
- View the buggy code and understand its context

## 2. Analyze Root Cause
- Trace through the failing test execution path
- Identify why the current implementation produces incorrect behavior

## 3. Design the Fix
- Plan minimal changes that address the root cause
- Consider impact on other parts of the codebase
- Ensure the fix doesn’t break existing functionality

## 4. Implement & Verify
- **Simple changes**: Use `sed -i` for straightforward replacements
- **Complex changes**: Use file reconstruction with `head`/`tail` when `sed` becomes too complex
- Test immediately: `defects4j test -r` (compiles and tests)
- If tests still fail: analyze error output and refine the fix
- Repeat until all tests pass

## 5. Multi-Hunk Strategy
For bugs spanning multiple locations:
- Understand the relationship between all fault locations
- Fix locations in logical order (dependencies first)
- Fix all related locations before testing (they often depend on each other)
- Verify all locations work together with `defects4j test -r`

# Success Criteria
- All failing tests pass: `defects4j test -r` shows no failures

**When all tests pass, signal completion with**: `echo COMPLETE_TASK_AND_SUBMIT_FINAL_OUTPUT`

Remember: You’re not just writing code - you’re debugging and fixing existing systems. Think like a detective: gather evidence, form hypotheses, test them systematically.

{% if has_blame_info %}

# Historical Context Available
```

```
You have access to git blame analysis showing how this code evolved. Use this context to:
- Understand previous changes and their rationale
- Identify patterns in how similar bugs were fixed
- Learn from code evolution and avoid regression
- Recognize architectural relationships and dependencies

Pay special attention to historical context - it often reveals the "why" behind code decisions.
{% endif %}
```