TraceGate: Policy-Based Disclosure of Structured Crash Evidence for LLM-Assisted Debugging

Nicolas Schuler[†], Balint Mate[*], Vincenzo Scotti[†] Raffaela Mirandola[†]
[†]*KASTEL, Karlsruhe Institute of Technology*, Karlsruhe, Germany
nicolas.schuler@kit.edu, vincenzo.scotti@kit.edu, raffaela.mirandola@kit.edu
[*]*FZI Research Center for Information Technology*, Karlsruhe, Germany
mate@fzi.de

***Abstract*—Large Language Models (LLMs) are increasingly used in automated debugging and repair. However, most current systems treat failure-time evidence in one of two ways: as a fixed prompt artifact or as the result of an open-ended, model-driven investigation. This misses a fundamental opportunity. In LLM-assisted debugging, the ability to inspect and control what information is made available when a program fails should be treated as a first-class control problem. We present TraceGate, an adaptation layer that sits between failure execution and patch generation. TraceGate captures structured crash snapshots and adaptively decides which evidence to disclose, whether to invoke diagnostic helpers, and when to escalate budget for repair. These decisions are made by lightweight online policies conditioned on failure characteristics, repair history, model identity, and remaining budget, without changing the underlying repair backend. We evaluate TraceGate by augmenting representative LLM-based repair approaches across multiple benchmarks and model sizes. Our results demonstrate that policy-controlled evidence disclosure recovers additional correct repairs in previously unsolved cases while improving the success-cost trade-off in most evaluated settings. Overall, TraceGate shows that rethinking debugging through controlled observability, rather than relying solely on stronger models or larger prompts, can make LLM-assisted repair more effective, efficient and controllable.**

***Index Terms*—Automatic Program Repair, Code Generation, Debugging, LLM**

I. Introduction

The software development life cycle is undergoing significant reshaping, affecting all stages in response to improvements driven by Large Language Models (LLMs) [1]. Code creation and validation (including automated repair workflows) are the most affected, especially due to the spread of *coding agents* [2], [3]. Yet, the generated code frequently fails tests, crashes at runtime, or contains subtle logic errors that require iterative debugging [4]–[8]. In these repair settings, success depends not only on generating a candidate patch but also on what failure-time information is exposed to the LLM after execution fails [8]. For LLM-based Autonomous Program Repair (APR), observability is therefore a core systems design decision rather than a peripheral implementation detail.

Existing repair frameworks generally employ one of two strategies. The first exposes a fixed package of *failure evidence*, such as compiler diagnostics, stack traces, or runtime summaries [4]–[6], [9]. The second lets the LLM conduct an *open-ended investigation* over multiple iterations, deciding when to inspect state, invoke tools, or collect additional traces [7], [8], [10]–[12]. This difference matters because the value of debugging evidence is non-monotonic: too little information forces the model to guess, whereas too much or poorly chosen information increases token cost, expands context, and can weaken reasoning rather than improve it [13]–[16]. Moreover, what constitutes useful evidence depends on (i) the model, (ii) the repair scaffold, and (iii) the remaining context and monetary budget [13], [17]–[20]. Thus, the focus shifts from "how to expose more failure information" to "how to disclose the most useful failure-time evidence under cost constraints." When neither evidence nor budget is externally governed, repair loops can fall into repetitive, low-yield behaviors.

In this work, we present *TraceGate*, an adaptation layer for LLM-based debugging and repair that sits between failure execution and patch generation. TraceGate captures a structured crash snapshot and makes three decisions that existing repair loops typically set fix in advance or leave implicit: (1) what evidence to disclose to the patch generator, (2) whether to invoke additional diagnostic helpers before repair, and (3) whether to escalate the budget after failed attempts. Rather than replacing the underlying repair backend, tool suite, or prompting scaffold, TraceGate extends these components. It adds controls over the failure context and resource budget. We built this extension with an LLM-centered process in mind, rather than a human-centered one in which the LLM guides human experts.

We perform an empirical evaluation comparing the effect of augmenting existing LLM-based repair tools and frameworks with TraceGate using existing code repair benchmarks. The results demonstrate that TraceGate consistently recovers additional correct repairs from residual failures left unsolved by the underlying backends. The strongest gains appear when more capable repair scaffolds can exploit better failure-time evidence. Some of these gains come at comparable or even lower token cost.

With our work, we make the following contributions:

- TraceGate: an adaptation layer for LLM-based debugging and repair that treats failure-time observability and repair budget as explicit control variables.

This work was supported by funding from the pilot program Core Informatics at KIT (KiKIT) of the Helmholtz Association (HGF) and KASTEL Security Research Labs, Karlsruhe.

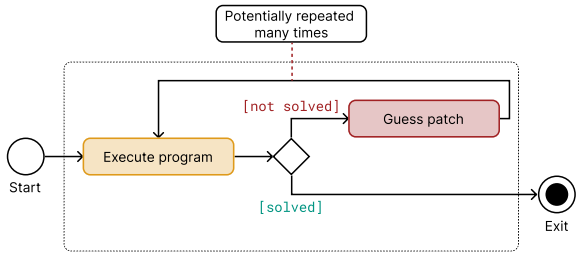

Fig. 1. Unconstrained LLM-based repair loops resulting from *LLM Roulette*.

- A lightweight online-learning policy mechanism for adaptive evidence disclosure, helper invocation, and budget escalation conditioned on failure characteristics, repair history, model identity, and remaining budget.
- An empirical study across four Python repair benchmarks, five repair backends, and three model sizes that measures residual repair recovery and the success-cost trade-off under adaptive observability control.

The remainder of this paper is organized as follows. Section II frames the problem of APR. Section III examines prior work on evidence disclosure, agentic debugging, and adaptive cost control. Section IV motivates and positions our contribution. Section V presents the TraceGate architecture and adaptation loop. Section VI describes the experimental setup and Section VII reports the main results. Section VIII interprets the findings, and Section IX discusses threats to validity. Finally, Section X concludes the paper, summarizing the contribution and suggesting possible future extensions.

II. Background

Autonomous Program Repair aims to automatically generate patches for software bugs. At a high level, APR approaches take an *input program* P along with *evidence* e indicating that the program fails (e.g., failing test cases, output of code analyzers), and produce a *patch* p that fixes the bug when applied to P, without negatively influencing other correct functionality [21]. How the fault was localized is orthogonal to repair and not directly relevant here. APR classically relies on techniques such as symbolic execution, evolutionary computation, machine learning, or more recently, generative AI [6], [21], [22]. LLMs in particular have proven to be well suited for the generation of patches for APR. In LLM-based APR, the *repair backend* M, with $M : (P,e) \mapsto p$, produces the patch. e includes source code, stack traces, input/output values, runtime state, or prior patches. However, the content of e comes at a cost. Each artifact consumes tokens from the *budget state* B, and oversized or poorly chosen contexts degrade repair quality rather than improving it (see Section IV). What to include in e while considering B is therefore a primary design decision and the main topic of this work.

A further challenge in LLM-based APR is frequently described as *Code Roulette*, which refers to the high variance in outputs produced by stochastic decoding strategies even when prompted with identical inputs, as well as to the high variance in outputs produced in response to slight changes in the prompt [23], [24]. Recent work shows LLMs can generate very different patches across runs, with correctness varying as sampling parameters like temperature or nucleus threshold change [2], [25]. This variability complicates evaluation and often necessitates repeated sampling or ranking strategies to identify plausible fixes. We distinguish between two related but distinct sources of variability: *Code Roulette*, which refers to patch-level stochasticity, and *LLM Roulette*, which denotes unstable repair processes driven by this stochasticity. While *Code Roulette* describes the variance in the generated code itself, *LLM Roulette* refers to the unstable repair dynamics that arise when such variability is embedded in an iterative repair process. In this setting, even small differences in early generations can alter subsequent execution states, observed evidence of failure, and follow-up prompts, thereby steering the repair process onto divergent trajectories. A common consequence is that the system becomes trapped in an unproductive and potentially never-ending repair loop, as depicted in Fig. 1.

A related but distinct failure mode is *repair degeneracy*. Whereas *Code Roulette* and *LLM Roulette* capture excessive variability and unstable repair trajectories, repair degeneracy describes the opposite problem: the tendency of models to collapse onto near-duplicate patches or syntactic and semantic loops, particularly when prior failed attempts are fed back into the prompt. Similar issues appear in neural text generation [26] and extend to code, reducing patch diversity and wasting budget B without aiding repair.

III. Related Work

In this section, we review related work along the following lines: LLM-based repair systems with predefined exposure to failure evidence, agentic debugging systems that let the model gather evidence on demand, and adaptive orchestration methods that allocate context, tools, or compute under budget constraints.

A. Evidence Selection for LLM-Based Repair

A central design choice in LLM-based repair is which artifacts from a failing run are serialized into the next prompt. This choice is consequential rather than cosmetic: prior work shows that both local context size [27] and task formulation [28] materially affect repair success.

Most existing systems rely on fixed evidence packages applied uniformly across all failure types. Self-DEBUGGING and SelfAPR feed back execution outcomes, compiler diagnostics, or test results to drive iterative repair [4], [5]. INTERVENOR rewrites compiler bug reports into its Chain-of-Repair scaffold to counter degeneration from unguided self-reflection [6]. LDB and DynaFix go further by exposing structured runtime artifacts such as variable values, control-flow paths, and call-stack information [9], [29]. These systems differ in their representation but not in their controls: the evidence schema is selected by the method designer and then applied uniformly across bugs and attempts.

More recent work begins to vary the evidence across instances. MANIPLE formalizes the fact-selection problem

and uses an offline-learned classifier to choose bug-specific prompt facts [30]. Ehsani et al. study layered context injection that escalates from bug-local information to repository- and project-level knowledge [31]. Haque et al. compare different trace representations and show that richer runtime traces are not uniformly better, reinforcing the non-monotonic value of debugging evidence [32]. This line is closest to TraceGate, but the adaptation is still chosen offline at design time, follows a fixed escalation schedule, or is applied once per task rather than being updated per attempt under an explicit budget.

B. Agentic Investigation and Tool Use in Debugging

An alternative to predefining the disclosure strategy is to let the LLM gather evidence on demand.

AutoSD [10] emulates scientific debugging: it formulates hypotheses, runs debugging experiments, and iteratively observes results before generating a patch. ChatDBG [11] and InspectCoder [12] integrate an LLM directly into debuggers and dynamic analysis tools, enabling it to autonomously issue commands and inspect the program state. TraceCoder [8] introduces a multi-agent pipeline in which an instrumentation agent inserts probes, a reasoning agent analyzes the resulting traces, and a repair agent produces patches. An open-ended, autonomous agent workflow that incorporates tools for reading and searching code, executing tests, performing fault localization, and editing files is presented in RepairAgent [7]. RepairAgent is especially suitable for interacting with larger codebases due to its repository-level context management and tool support.

These approaches demonstrate that richer, on-demand evidence gathering can improve APR. However, in the current landscape, the LLM controls which tools it uses and which evidence it acquires, with no external governance over the scope, cost, or relevance of the evidence it discloses. This lack of external governance creates the conditions for the *LLM Roulette* problem introduced in Section II: tool overuse, rapidly growing context, and weakened reasoning.

IV. Motivation

In this section, we clarify the motivations for adopting adaptive control and position our contribution in this context, comparing it with related work.

A. Factors Motivating Adaptive Control

Hereafter, we discuss four reasons motivating a LLM-based repair process to consider failure characteristics, models, and previous repair attempts.

a) Context degradation: Long-context studies consistently show that more input does not monotonically improve LLM performance. Models exhibit U-shaped attention, favoring information at the beginning and end of the prompt while degrading in the middle [15]. Even when relevant information is retrievable, longer contexts can harm performance [14], and agentic settings exacerbate this effect as context length grows [33]. Contextual distractors further induce severe performance drops, amplified by multi-turn and agentic workflows [16]. These findings indicate that evidence disclosure should be incremental and failure-specific, rather than assuming that a richer context is always beneficial.

b) Tool and turn control: Unconstrained agentic repair loops tend to overuse tools and turns, even when sufficient information is already available [13], [34], [35]. Prior work shows that externally managing tool invocation and interaction depth can both reduce cost and improve success rates [13], [34], [35]. Practical systems mitigate this behavior through turn compression or context summarization [36], [37]. Collectively, these results argue that evidence acquisition should be externally governed rather than left entirely to model-driven exploration.

c) Budget-aware allocation: Iterative repair with large models incurs substantial computational and monetary costs, motivating budget-aware agent design [13], [19], [20], [38]. Budget-adaptive allocation has been explored across models, prompts, solvers, and reasoning effort in related settings [39]–[43]. However, these methods treat budget allocation independently of debugging evidence. In practice, evidence disclosure and budget are tightly coupled: richer evidence consumes more tokens, while strict budgets constrain observability. This coupling remains largely unexplored.

d) Cross-model heterogeneity: Recent benchmarks reveal substantial heterogeneity across frontier models in tool use, reasoning, and coding behavior [18]. These differences are further shaped by prompting strategies, tool interfaces, and orchestration scaffolds. As a result, a disclosure strategy that is effective for one model may degrade performance in another. Evidence policies should therefore condition on the backend model and repair scaffold, rather than assuming a model-agnostic mapping from failure to evidence.

B. Adaptive Evidence-Budget Control

Prior work converges on a common insight: adaptive allocation of limited resources outperforms uniform allocation across context, models, turns, or prompts. TraceGate adopts this control perspective, but applies it to two *coupled* resources: the structured debugging evidence disclosed to the repair backend and the token budget, conditioned on failure characteristics and model identity.

While existing approaches adapt either evidence [30], [31] or budget [19], [38], none jointly control both. Agentic systems can acquire richer evidence [7], [8], [10], but do so without explicit governance over disclosure scope or repair spending. TraceGate couples evidence selection and budget control instead through configurable online learning policies, while remaining orthogonal and composable with existing repair backends. Table I summarizes this positioning.

V. Approach

TraceGate augments a standard LLM-assisted repair loop with adaptive control over *failure-time observability*. Before each repair attempt, it decides which failure evidence to disclose and whether to retain or escalate the current budget.

TABLE I
POSITIONING OF TRACEGATE RELATIVE TO RELATED WORK.

Direction	Examples	Exposed runtime state	Fixed evidence	LLM-driven gathering	Adaptive evidence	Budget-aware	Composable
Compiler-guided interactive self-repair	Self-DEBUGGING [4], Self-APR [5], INTERVENOR [6]		✓				
Static runtime-context prompting	LDB [9]	✓	✓				
Live debugger-centered inspection	AutoSD [10], ChatDBG [11], InspectCoder [12]	✓		✓			
Trace-driven multi-agent debugging	TraceCoder [8]	✓		✓	(✓)		
Learned fact selection	MANIPLE [30]	✓	(✓)		✓		
Layered context escalation	Ehsani et al. [31]	✓	(✓)		✓		
Adaptive evidence-budget control	**TraceGate (ours)**	✓	(✓)	(✓)	✓	✓	✓

✓= supported; (✓) = partially or conditionally supported. Exposed runtime state: exposes runtime variables, traces, or control flow. Fixed evidence: provides a predetermined evidence package. LLM-driven gathering: LLM can gather additional evidence during repair. Adaptive evidence: varies the evidence disclosed based on the bug or attempt. Budget-aware: considers the token or cost budget. Composable: composable with existing repair backends.

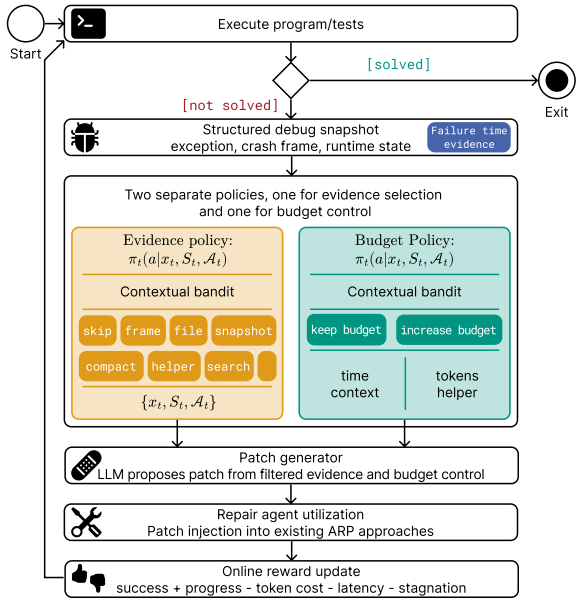

Fig. 2. TraceGate End-to-End Workflow.

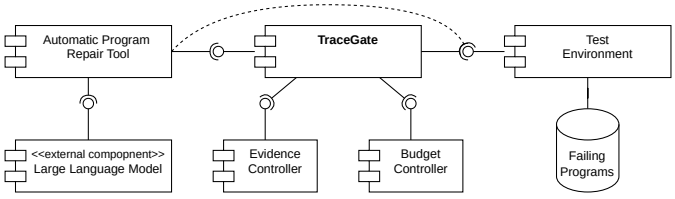

Fig. 3. TraceGate-enhanced APR component architecture.

A. Overview

TraceGate operates between failed execution and patch generation. Given a failing program or test case, it first captures a structured crash snapshot that includes failure-time state, such as exception information, crash-local execution context, and compact runtime metadata. This snapshot serves as the canonical evidence source for subsequent repair attempts. The overall process is visualized in Fig. 2.

After the initial failed execution, TraceGate enters a bounded retry loop. At each attempt, it builds a state representation of the current repair situation and makes two decisions. A *budget policy* determines whether to retain the current budget profile or escalate available resources, such as token budget. An *evidence policy* determines what failure-time evidence to disclose to the patch generator. Depending on the selected evidence action, TraceGate may reveal a lightweight snapshot view, request helper-mediated context, or proceed with a minimal evidence package.

The selected evidence and budget are then passed to the underlying patch generator (e.g., INTERVENOR [6]), which proposes a candidate patch. The patch is validated through execution, and the observed outcome is converted into an online reward signal to update the adaptive policies. If repeated failures do not yield materially new evidence, TraceGate can skip patch generation for that iteration to avoid a low-yield retry. This loop terminates once the task is solved or the retry budget is exhausted.

B. Architecture

At the architectural level, TraceGate wraps an existing repair loop with an explicit evidence-and-budget controller (see Fig. 3). It leaves the underlying patch generator, tool suite, and prompting scaffold unchanged.

The architecture consists of five components. First, a *snapshot collector* captures and normalizes failure-time state after unsuccessful execution. Second, a *state builder* summarizes the current repair situation from the failure outcome, snapshot-derived diagnostics, repair history, model and scaffold context, and remaining budget. Third, two adaptive controllers operate on this state: a budget controller decides whether to retain or increase the current budget profile, and an evidence controller decides which evidence to materialize for the next repair attempt. Fourth, an *evidence materializer* converts the selected evidence action into the concrete context shown to the patch generator, ranging from

compact snapshot views to helper-mediated context acquisition. Finally, an *outcome assessor* validates candidate patches, computes reward signals, and updates the online policies.

This decomposition separates *collecting* debugging evidence from *deciding how much of it to disclose*. It also keeps TraceGate orthogonal to existing APR frameworks: the same repair backend can be paired with different policies without modifying the backend itself.

C. Adaptation Mechanism

We model TraceGate's adaptive controllers as *contextual bandits*. At each repair attempt, the system observes a context vector summarizing the failure state, structured snapshot, repair history, and remaining budget, then selects actions for budget control and evidence disclosure. This fits the contextual bandit setting [44], [45], and in our implementation, we use a lightweight linear method [46]. The key challenge is *attempt-local exploration under partial feedback*: after each failure, TraceGate must decide what evidence to reveal and how much budget to spend, but it never observes the counterfactual outcomes of the actions it did not take. The problem is therefore not long-horizon planning, as in full reinforcement learning, but online action selection from feature-based failure contexts. In practice, lightweight linear contextual-bandit methods are attractive here because they support efficient online updates, explicit exploration, and low-overhead action selection in discrete action spaces.

Algorithm 1 summarizes the core adaptive control flow of TraceGate. It abstracts over engineering details, such as delayed credit assignment, while preserving the key method-level decisions: building a contextual state, selecting budget and evidence actions, optionally skipping redundant retries, validating candidate patches, and updating both policies online from observed outcomes.

Formally, let T denote the maximum number of repair attempts after the initial failing run. TraceGate maintains four pieces of mutable loop state: the current failure outcome o, the most recent structured snapshot s, the budget state B, and the repair history H (initialization in lines 1-7). At attempt t, these are summarized into a context vector

$$x_t = \phi(o,s,B,H),$$

where $\phi(\cdot)$ aggregates several feature families, including failure characteristics, snapshot diagnostics, repair-history signals, prior helper behavior, model and scaffold context, and remaining budget (line 8). For readability, we describe these feature families only at a high level in the main text and defer the full feature inventory to the implementation. Given x_t, TraceGate applies two contextual-bandit policies. The budget controller

$$a_t^B = \pi_B(x_t)$$

selects whether to retain the current budget profile or increase the resources available to the current attempt (line 9). The evidence controller

$$a_t^E = \pi_E(x_t)$$

selects what failure-time evidence should be disclosed (line 11). The resulting evidence is then materialized as

$$e_t = \mathcal{M}(a_t^E, s),$$

where $\mathcal{M}$ converts the abstract evidence action into the concrete context shown to the patch generator (line 12). Depending on the selected evidence action, this may range from lightweight snapshot views to helper-mediated context acquisition.

Before invoking the repair backend, TraceGate applies a retry gate, denoted by SHOULDSKIPPATCH (line 13). This gate suppresses patch generation when the current failure effectively repeats a previous one, and the selected evidence adds no materially new information. In that case, another repair attempt is unlikely to be useful, so TraceGate skips patch generation and proceeds to the next round (lines 14-17). Otherwise, it passes the selected evidence and current budget profile to the repair backend to obtain a candidate patch, which is then validated by execution. The resulting outcome o' is converted into an online reward

$$r_t = \mathcal{R}(o', H),$$

that favors successful repairs and meaningful intermediate progress while penalizing wasted resources and repeated low-yield behavior (lines 18-20). At a high level, the reward combines task outcome and progress signals with costs such as token usage, latency, and stagnation. Both adaptive controllers are then updated online from this reward, allowing TraceGate to gradually learn which evidence and budget actions are effective (line 21).

At each step of the process, o,s,B,H are updated to maintain the current repair state (lines 10 and 24-26). Whenever a valid solution is found or the maximum number of iterations is reached, the algorithm stops (lines 22-23 and 27).

VI. EVALUATION

We evaluate TraceGate as an augmentation layer for existing LLM-based repair backends. First, this section defines the research questions, then describes the experimental setup and baseline context.

A. Objectives

We organize the evaluation around three research questions:

RQ1: *How effectively does TraceGate recover correct repairs on residual failures left unsolved by existing repair backends?*
This question tests whether adaptive disclosure of structured crash evidence can recover additional correct repairs after the underlying backend has exhausted its bounded repair attempts.

RQ2: *What token cost does TraceGate add, and when does it improve the recovery-cost trade-off?*
This question measures the token cost of augmentation and asks in which settings added recovery comes at neutral, lower, or acceptable overhead relative to the underlying backend on the same residual subset.

Algorithm 1: TraceGate adaptive repair loop

Input: failing task x,
repair backend M, maximum repair attempts T
Output: successful patch or failure

$o \leftarrow \textsc{Execute}(x)$;
if *o is successful* **then**
  return `no_patch_needed`;
$s \leftarrow \textsc{CaptureSnapshot}(o)$;
$B \leftarrow \textsc{InitializeBudget}()$;
$H \leftarrow \emptyset$;
for $t \leftarrow 1$ **to** T **do**
  $x_t \leftarrow \phi(o,s,B,H)$;
  $a^B \leftarrow \pi_B(x_t)$;
  $B \leftarrow \textsc{ApplyBudget}(B,a^B)$;
  $a^E \leftarrow \pi_E(x_t)$;
  $e \leftarrow \mathcal{M}(a^E,s)$;
  if *SHOULDSKIPPATCH(o, e, H)* **then**
    $r \leftarrow \mathcal{R}(\texttt{skip},H)$;
    $\textsc{UpdatePolicies}(x_t,a^B,a^E,r)$;
    $H \leftarrow H \cup \{(x_t,a^B,a^E,\texttt{skip},r)\}$;
    continue;
  $p \leftarrow \textsc{GeneratePatch}(M,e,B)$;
  $o' \leftarrow \textsc{ValidatePatch}(x,p)$;
  $r \leftarrow \mathcal{R}(o',H)$;
  $\textsc{UpdatePolicies}(x_t,a^B,a^E,r)$;
  if *o' is successful* **then**
    return p;
  $o \leftarrow o'$;
  $s \leftarrow \textsc{CaptureSnapshot}(o)$;
  $H \leftarrow H \cup \{(x_t,a^B,a^E,o,r)\}$;
return `failure`;

RQ3: *How do TraceGate's learned evidence-disclosure and budget policies adapt across models and repair backends?*
This question tests whether the learned controller settles on a single static strategy or instead changes its evidence and budget decisions across settings.

Baseline results establish the residual context for interpretation, recovery results answer RQ1 and RQ2, adaptation results address RQ3, and ablations provide robustness checks for the evaluation design.

B. Experimental Setup

All experiments were run on a single workstation with a Ryzen Threadripper 9970X CPU and an NVIDIA RTX PRO 6000 GPU with 96 GB VRAM. We deployed all models locally using `llama.cpp` and used 6-bit-quantized Qwen3.5 checkpoints [47] in three sizes: 4B, 9B, and 35B; reasoning mode was disabled throughout, and both model serving and benchmark execution were containerized. For reproducibility, we followed the recommended `llama.cpp` deployment setup [48] and reused the original baseline implementations where available, modifying only the execution wrapper needed to insert TraceGate as an external control layer.

TABLE II
BENCHMARK SIZES FOR OUR EXPERIMENTS.

Benchmark	Total Cases	Ablation Samples
ConDefects [49]	1,267	64
DebugBench [50]	328	56
HumanEvalPack [51]	164	48
Hand-crafted	69	35

1) Benchmarks: We evaluate on four Python repair benchmarks used in prior APR work, including one hand-crafted set; Table II summarizes their sizes. Some source benchmarks are multilingual, so we restrict the study to Python to keep the execution environment, tooling, and prompt format consistent across tasks. Each task provides a buggy program, an associated failing test, and execution-derived failure information, and the repair objective is to generate a patch that makes the provided test suite pass. In addition to existing benchmarks, we propose a handcrafted benchmark designed to test solving capabilities for bugs related to asynchronous code execution. For the robustness analyses in Section VII-D, we also define a smaller ablation subset from the same benchmarks, sized to support repeated analyses at manageable cost while targeting a precision of approximately ± 12 percentage point (pp) at 95% confidence [52].

2) Repair Backends: We compare TraceGate against five representative repair loops spanning the design regimes discussed in Section III: minimal traceback prompting, stronger prompting scaffolds, decomposition-based generation, iterative repair from external feedback, and agentic trace-driven debugging.

- **Simple**: A minimal baseline that prompts the model with the failure information without additional scaffolding.
- **CoT** [53]: The same baseline augmented with chain-of-thought prompting.
- **LambdaRLM** [54]: A decomposition-oriented reasoning scaffold.
- **INTERVENOR** [6]: An iterative repair backend built around the authors' Chain-of-Repair mechanism.
- **TraceCoder** [8]: A trace-driven multi-agent repair pipeline.

3) Protocol and Metrics: Each task begins with a failed execution at $t=0$. This run is not counted as a repair attempt; it only exposes the original failure and, for TraceGate, collects the initial structured snapshot. In the base condition, each repair backend then receives up to two bounded repair attempts, $t \in \{1,2\}$, so *Solved@1* counts tasks solved on the first repair attempt, and *Solved@2* counts tasks solved cumulatively within two attempts.

Our primary comparison evaluates TraceGate as a post-failure augmentation layer rather than as a stand-alone, budget-matched replacement for the repair backend. For each model, backend, and dataset, we first run the unmodified backend on the full benchmark, then rerun TraceGate only on the residual

subset left unsolved after the backend's two repair attempts. The reported Δtok values measure the total change in tokens for the augmented run on the same residual subset, relative to the underlying backend, across identical tasks. Policies are trained online within each dataset-backend-model run and reset between runs to prevent information leakage across experimental conditions. Because each backend leaves a different residual subset, comparisons across rows reflect both augmentation behavior and residual-task composition, so we interpret the gains as run-level marginal improvements rather than exact set-wise quantities. Alongside *Solved@1*, *Solved@2*, and token consumption, we log the number of model calls, latency, and failure categories as secondary diagnostic measures.

VII. Results

In this section, we answer RQ1–RQ3 through recovery, policy-behavior, and robustness analyses.

A. Baseline Results

Table III establishes the baseline context for the residual recovery analysis and clarifies what kind of residual headroom TraceGate inherits from each backend. INTERVENOR is the strongest overall baseline in our setting, achieving the best aggregate final total, while the traceback-only baseline is usually the cheapest in token consumption. Larger models generally improve solved rates across backends, and these gains exceed the incremental benefit of a second repair attempt within a fixed setting: moving from 4B to 9B increases solved cases by 10.6% while reducing token consumption by 1.6%, and moving from 9B to 35B adds another 9.8% solved cases while reducing tokens by 10.1%. This pattern suggests that baseline performance is more sensitive to model capacity than to allowing one additional retry within the same repair scaffold. Most solutions are obtained at $t=1$; on average, the additional cases solved at $t=2$ account for only about 8% of final solved cases, which motivates evaluating TraceGate on the harder residual subset rather than as a replacement for the base repair loop. Put differently, the residual tasks passed to TraceGate are not only fewer than those in the original benchmark cases but also systematically harder. The baselines also leave residual subsets of very different sizes, so the augmentation results should be read both as fractional recovery on the residual pool and as absolute lift in final solved rate. The repair-loop TraceCoder underperforms relative to the original paper's results in our setting, likely because our experiments use smaller local models than the substantially larger LLMs used in their original evaluation.

B. Recovery Results

RQ1 asks whether TraceGate can recover correct repairs on residual failures left unsolved by the underlying repair backends. Table III shows that it does so in all 15 evaluated model-backend settings, with an average residual recovery rate of 20.5%. The strongest fractional recoveries appear when TraceGate augments stronger repair backends: INTERVENOR recovers 28.4–35.9% of residual failures, LambdaRLM 19.9–30.3%, and TraceCoder 17.0–34.6%. These recoveries

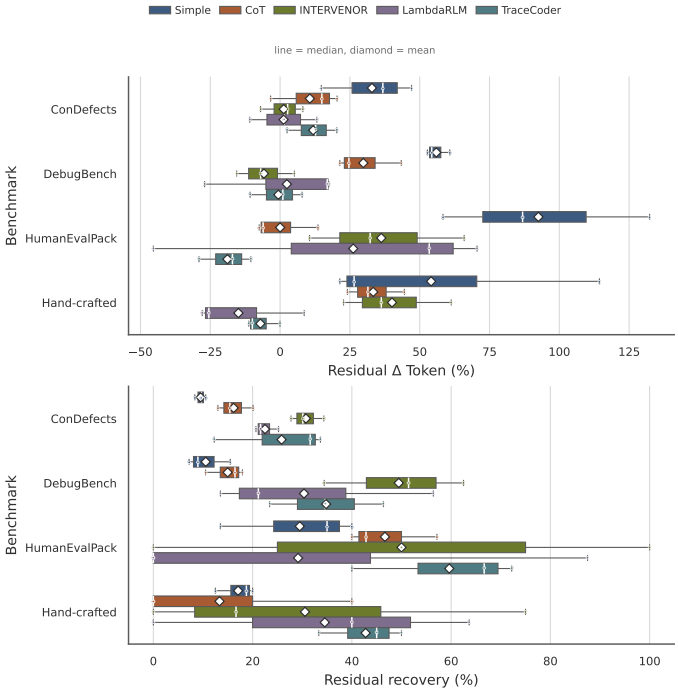

Fig. 4. Residual recovery vs. Δtoken aggregated over model sizes.

translate into substantial final *Solved@2* gains; for example, INTERVENOR improves from 66.2% to 75.8% at 4B, from 72.3% to 81.1% at 9B, and from 79.3% to 86.7% at 35B. This pattern distinguishes complementarity from residual headroom: stronger backends achieve the highest fractional recovery on their remaining failures, whereas backends with larger residual pools can still produce the largest absolute lift in final solved rate. TraceCoder illustrates the latter most clearly, improving final *Solved@2* by $+11.1$, $+14.8$, and $+12.4$ pp at 4B, 9B, and 35B, respectively. The gains are also front-loaded: across all settings, about 67.3% of recovered cases appear on the first recovery attempt, rising to about 79.0% for INTERVENOR.

> **Answer to RQ1:** TraceGate consistently recovers additional correct repairs on residual failures across all evaluated settings; stronger backends show the highest fractional residual recovery, while weaker backends with larger residual pools can still produce the largest absolute solved-rate lift.

RQ2 asks what token cost this recovery adds, and the answer is strongly backend-dependent. As shown in Fig. 4, traceback and CoT define the clearest high-cost/low-recovery region, whereas the stronger scaffolds often recover more residual cases at modest overhead and sometimes at lower token cost than the underlying backend on the same residual subset. Three settings already improve on the underlying backend on both axes: LambdaRLM-4B (30.3% residual recovery at -12.1% tokens), CoT-9B (20.4% at -2.3% tokens), and INTERVENOR-35B (35.9% at -7.1% tokens). Three further settings recover 21.8–31.8% of residual failures at only $+2.3$–2.4% token overhead:

TABLE III
BASELINE *Solved@1*/*Solved@2* COUNTS AND TRACEGATE GAINS.

Model	Approach	ConDefects ($N=1267$)				DebugBench ($N=328$)				HumanEvalPack ($N=164$)				Hand-crafted ($N=69$)			
		Solved@1		Solved@2		Solved@1		Solved@2		Solved@1		Solved@2		Solved@1		Solved@2	
		Base	Trace Gate	Base	Trace Gate	Base	Trace Gate	Base	Trace Gate	Base	Trace Gate	Base	Trace Gate	Base	Trace Gate	Base	Trace Gate
Qwen3.5-4B	Simple	316	+46	370	+75	124	+7	172	+14	115	+2	127	+5	49	+3	53	+3
	CoT [53]	409	+60	483	+102	121	+16	176	+25	121	+4	143	+9	60	+0	62	+0
	LambdaRLM [54]	606	+98	708	+141	214	+40	250	+44	151	+4	156	+7	57	+4	58	+7
	INTERVENOR [6]	643	+123	719	+152	247	+15	267	+21	157	+1	162	+1	63	+1	63	+1
	TraceCoder [8]	152	+69	264	+123	159	+31	201	+44	104	+22	128	+26	46	+8	49	+9
Qwen3.5-9B	Simple	435	+48	509	+80	173	+1	217	+8	133	+3	144	+7	53	+2	59	+2
	CoT [53]	558	+85	635	+127	160	+10	211	+21	127	+5	150	+8	59	+0	66	+0
	LambdaRLM [54]	689	+66	793	+98	260	+3	291	+5	155	+0	160	+0	66	+0	66	+0
	INTERVENOR [6]	723	+131	799	+141	263	+11	293	+18	162	+1	162	+2	64	+0	67	+0
	TraceCoder [8]	388	+166	476	+250	238	+11	264	+15	154	+2	158	+4	61	+1	63	+2
Qwen3.5-35B	Simple	605	+29	697	+55	211	+1	257	+11	149	+1	159	+2	57	+1	61	+2
	CoT [53]	702	+48	800	+72	216	+1	271	+6	151	+1	159	+2	60	+1	64	+2
	LambdaRLM [54]	850	+46	943	+70	177	+18	224	+22	159	+0	162	+0	54	+3	59	+4
	INTERVENOR [6]	849	+99	909	+123	300	+6	312	+10	161	+0	163	+0	62	+3	65	+3
	TraceCoder [8]	508	+132	661	+204	276	+10	287	+19	157	+2	159	+2	64	+2	65	+2

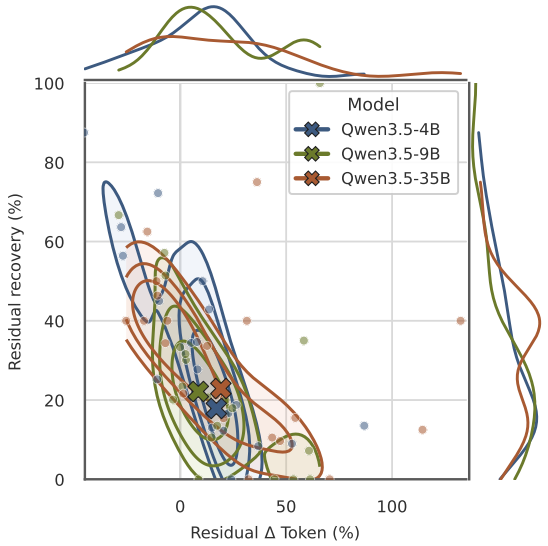

Fig. 5. Residual recovery vs. Δtoken, by model size. × indicates average.

TABLE IV
ACTION-RECOVERY ASSOCIATIONS BY MODEL SIZE. POSITIVE VALUES INDICATE OVERREPRESENTATION AMONG RECOVERED CASES; NEGATIVE VALUES INDICATE UNDERREPRESENTATION.

Evidence actions			
Action	**4B**	**9B**	**35B**
invoke	+2.6	+2.5	-5.4
frame+locals	-2.9	-0.7	+2.1
snap-lite	-1.4	-0.4	+1.6
snap-full	+1.8	-0.8	+4.5
skip	+1.6	+1.3	-0.5
file	+0.9	+0.4	-0.3
frame	+0.2	-0.9	-1.1
search	-0.9	-0.5	-0.4
compact	-1.9	-0.9	-0.5
retry-json	+0.0	+0.0	-0.0

Budget actions			
Action	**4B**	**9B**	**35B**
keep	+2.0	-0.0	+6.7
increase	-2.0	+0.0	-6.7

Answer to RQ2: The token cost of recovery depends strongly on the backend: several settings already improve both recovery and token efficiency, whereas the least favorable regime is simple augmentation with high overhead for limited residual recovery.

INTERVENOR-9B, TraceCoder-9B, and LambdaRLM-35B. By contrast, Simple-4B and Simple-35B require +37.8% and +47.7% more tokens for only 8.8% and 10.6% residual recovery, respectively. Fig. 5 shows the same trade-off broken down by model size. The joint density appears well-behaved, with smooth and largely overlapping high-density regions across models. In particular, the distributions align closely around their modes, suggesting a stable and comparatively predictable residual trade-off rather than strongly fragmented or erratic behavior. Furthermore, in all model sizes, the average recovery remains strictly greater than the residual Δtoken consumption.

C. Adaptation Results

RQ3 asks whether the learned evidence-disclosure and budget policies adapt across settings or converge to a single static strategy; the evidence supports the former. Even within the Qwen family, the action associations differ substantially across model sizes, so a fixed disclosure policy would fit some settings and fit others poorly. Table IV shows, for example, that `invoke` is positively associated with recovery for the 4B and 9B models (+2.6 and +2.5) but negatively associated for 35B (−5.4), whereas `snap-full` becomes most favorable at 35B (+4.5) after being only mildly helpful or negative at smaller sizes. This lack of monotonicity matters because the controller's useful behavior does not simply scale with model size. The

broader recovery results are consistent with the same interaction pattern across backends, with INTERVENOR and TraceCoder benefiting more consistently from scale than LambdaRLM.

Budget behavior also varies across model sizes (see Table IV). Keeping the current budget is mildly favorable at 4B (+2.0), neutral at 9B, and strongly positive at 35B (+6.7), while increasing the budget is neutral or unfavorable in the same settings. These observations suggest that budget escalation is not a universally good fallback; its value depends on the repair context and the model-backend combination.

The learned policy also concentrates decisions on a small subset of evidence actions rather than spreading them uniformly across the action space. Across our runs, full helper invocation accounts for 28.7% of evidence decisions, 54.8% are routed to lighter actions, and 16.5% are skipped. Fig. 6 complements the per-action summary by showing that the controller relies mainly on a few evidence actions while using budget increases sparingly across model sizes. This selectivity strengthens the paper's main claim: TraceGate is not only adaptive, but also selective in that it avoids making helper-heavy evidence acquisition the default behavior.

Answer to RQ3: TraceGate's learned controller does not collapse to a single best disclosure strategy; evidence and budget choices vary across settings, and the policy uses helper-heavy actions selectively rather than as a blanket default.

D. Ablations

The ablation evidence in this section serves as robustness checks for the evaluation design rather than as a second efficacy claim.

1) Baseline Stability: To assess how stable the residual subset is under backend stochasticity, we reran the same benchmark-backend configuration three additional times, yielding a total of four runs. Across these four runs, overall *Solved@2* remains stable at 81.8% ± 1.3 pp, while 5.7% ± 1.4pp of cases transition from unsolved to solved between runs. The residual subset is therefore not deterministic, but the observed variation is bounded relative to the recovery gains reported above. Accordingly, we interpret the main augmentation results as marginal improvements under mild benchmark noise rather than exact set-wise invariants. This variation is also smaller than the strongest absolute solved-rate lifts reported in Table III, which range up to +14.8 pp, reinforcing the importance of reporting variance explicitly in LLM-based software engineering evaluation [55].

2) Order Sensitivity of Online Adaptation: Across five order-randomized adaptive reruns, aggregate performance remained stable, with a mean solve rate of 83.3% (range 81.8–87.0%, standard deviation 1.9 pp), and four of five runs falling within four solved cases of each other. Budget-policy behavior was highly consistent across runs, with 86.6% pairwise same-action agreement and action-distribution cosine similarity of 0.998, choosing `keep_budget` in 86.2–95.1% of decisions.

Helper-action behavior was more variable at the exact-action level (24.0% pairwise exact-action agreement), but it remained within the same broad evidence-selection regime: snapshot-family actions were dominant in every run (32.8–44.9% of helper decisions), while `search_only` and `frame_only` remained rare (2.9–5.1% and 3.4–7.5%). Among the 149 cases solved in all five runs, only 3 used the same first-attempt helper action in every run, compared with 111 using the same first-attempt budget action. Overall, random ordering did not induce qualitatively different controller regimes; instead, it produced a stable budget policy and a diverse set of successful helper micro-policies, suggesting that adaptation remains beneficial even when exact helper choices vary.

TABLE V
TRACEGATE WITH DIFFERENT MODELS.

Model	Approach	Solved@1		Solved@2	
		Base	Trace Gate	Base	Trace Gate
Gemma-4-31B	INTERVENOR	1450	+73	1505	+90
GLM-4.7-Flash	INTERVENOR	902	+127	1048	+161

3) Fixed Evidence vs. Adaptive Policies: To isolate the effect of adaptation itself, we compared the adaptive controller against a static INTERVENOR variant that injects the full runtime snapshot together with deterministic failure-state evidence. This comparison uses the 203-case ablation set and averages the adaptive controller over five order-randomized reruns. The static variant solves 163/203 cases (80.3%), whereas the adaptive controller solves 169/203 cases on average (83.3%), a gain of 6 cases (3.0 pp). The efficiency gap is larger: the static variant consumes 2,994,881 total tokens, compared with 2,449,455 for the adaptive mean, corresponding to 22.3% higher token usage. In this ablation set, the benefit of TraceGate does not stem from always disclosing the richest available evidence; rather, adaptation improves final performance while avoiding the token cost of uniformly materializing the full snapshot.

4) Different Model Families: For comparison across other LLM families, we reran the full benchmark suite (see Table V) with INTERVENOR using Google's 31B Gemma-4 model [56] and Z.ai's 30B GLM-4.7-Flash model [57], both quantized to 6-bit. On Gemma-4, TraceGate recovered 27.9% of the residual cases, and 20.64% on GLM-4.7-Flash. The fixed evidence-package baseline solved only slightly fewer residual cases (26% and 21.28%), but required 19.4% and 14.58% more tokens, respectively. This result reinforces that TraceGate's benefit is not merely in recovering failures, but in doing so more efficiently through its joint orchestration of evidence, helper policy, and budget policy.

VIII. DISCUSSION

The central finding of the paper is that TraceGate is most useful as a residual-recovery layer rather than as a stand-alone repair backend. It is designed to act after a repair backend exhausts its bounded attempts, and the evaluation shows that it can recover an additional portion of the resulting residual failure

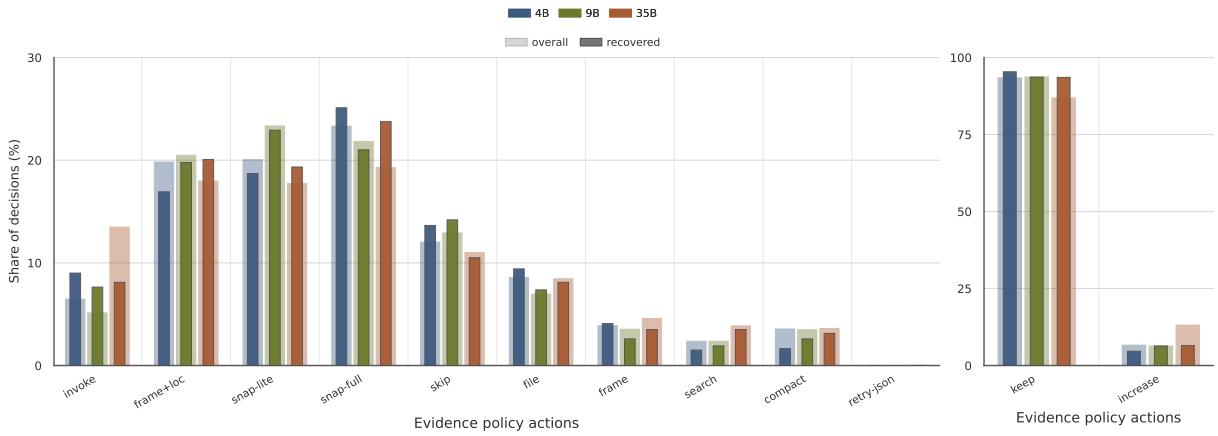

Fig. 6. Evidence and budget action mix across all attempts and recovered cases, by model size.

set. Stronger backends tend to achieve the highest fractional recovery on their smaller residual pools, while weaker backends can still produce larger absolute solved-rate lifts because they leave more residual headroom. Taken together, this pattern suggests that backend quality and adaptive evidence control are complementary rather than competing design choices.

The policy analysis suggests that the gain does not come from exposing more crash evidence by default. Instead, the useful behavior is selective: evidence-action preferences change across model sizes and repair backends, helper-heavy acquisition is concentrated on a subset of cases, and budget increases are not uniformly beneficial. This argues against both fixed evidence bundles and simple monotonic escalation heuristics. For LLM-based repair, failure-time observability is therefore better treated as a controllable resource than as a one-time prompt-design decision.

The success-cost trade-off is strongly backend-dependent. In several settings, TraceGate improves both recovery and token efficiency by avoiding unnecessary context growth and helper use through selective disclosure. In weaker traceback-style settings, however, additional evidence can still be expensive relative to the number of recovered cases. Adaptive observability is therefore most attractive when paired with repair backends that can convert targeted failure information into better patches, rather than as a blanket substitute for stronger repair scaffolds.

The current controller learns online within each dataset-backend-model run. This makes the design well-suited to repeated repair workloads, but it leaves open the extent to which policy knowledge can transfer across datasets, model families, and deployment environments. The most immediate next steps are warm-started or offline-trained controllers, richer state representations that capture longer repair history, and evaluations on larger multi-file debugging tasks where evidence selection and context cost are likely to matter even more.

IX. Threats to Validity

a) Construct validity: Our primary effectiveness metric is benchmark-level success on the provided test suites. As in prior APR work, this is an operational proxy for repair quality rather than a proof of full semantic correctness, so some recovered patches may still overfit the benchmark tests. We mitigate this threat by using established repair benchmarks with their original validation harnesses and by phrasing our claims in terms of additional benchmark cases recovered, rather than as universally correct repairs. A second construct threat concerns efficiency: we use token consumption as the main cost metric. Although we also log latency and model calls, the paper's efficiency claims should therefore be read as token-efficiency claims rather than as full deployment-cost analyses. Finally, the benchmark tasks may overlap with the pretraining corpora of modern code LLMs. We cannot rule out this possibility, but it is less likely to account for the paper's main result because every TraceGate variant is compared against the same underlying model and repair backend on matched tasks.

b) Internal validity: The main internal threat is that TraceGate learns online, so its behavior can depend on task order and on the residual subset produced by a stochastic backend run. We mitigate this by resetting the controller for each dataset-backend-model combination, running all methods in the same containerized environment, and reusing baseline implementations, except for the minimal wrapper required to insert TraceGate. We also report a stability check showing bounded but non-zero variation across repeated baseline runs. Still, because the residual subset is itself not deterministic, the reported gains should be interpreted as marginal improvements for a fixed run configuration rather than as exact case-wise invariants.

c) Conclusion validity: Our study prioritizes broad cross-setting coverage over dense repeated sampling. The main comparison spans 15 model-backend settings, but most full-benchmark results are based on single end-to-end runs, and the ablation analyses use a smaller subset for practical cost reasons. As a result, some per-setting differences may be sensitive to residual-subset size and run noise. We mitigate this by avoiding fine-grained causal claims, emphasizing directional patterns that recur across many settings, and reporting rerun variation where available. Stronger statistical conclusions would require repeated full-benchmark runs and paired confidence intervals for both recovery and token deltas.

d) External validity: Our evaluation covers four Python benchmarks and focuses on tasks with executable failure evidence, typically in small programs or otherwise localized repair settings. The findings may therefore not transfer directly to larger repositories, multi-file patches, other programming languages than Python, logic bugs without informative crash evidence, or human-in-the-loop debugging workflows. Similarly, we study only the Qwen3.5 model family in three sizes, served locally via `llama.cpp` with quantization and reasoning disabled. This isolates the effect of adaptive disclosure under controlled conditions, but it does not establish that the learned action preferences or success-cost trade-offs will remain unchanged for other model families (except our models in the ablations), closed-source APIs, or different inference stacks.

X. Conclusion

In this paper, we argued that failure-time observability in LLM-assisted repair should be treated as a control problem rather than as a fixed prompt-design choice or an unconstrained agent decision. We presented TraceGate, an adaptation layer that captures structured crash snapshots and learns online which evidence to disclose, whether to invoke diagnostic helpers, and when to retain or escalate the repair budget, without changing the underlying repair backend.

Across four Python repair benchmarks, five repair backends, and three Qwen3.5 model sizes, TraceGate recovered additional correct repairs in all 15 evaluated settings, averaging 20.5% recovery on the residual failures left unsolved by the underlying backends. The strongest relative improvements appeared when TraceGate was paired with already capable repair backends. Whereas, backends with larger unsolved repairs could still produce the largest absolute solved-rate improvement. Several settings improved both recovery and token efficiency simultaneously. These results suggest that adaptive disclosure of failure-time evidence is a practical way to make LLM-based repair more controllable and more cost-effective.

Future work should extend TraceGate to work with more complex, multi-step and multi-file debugging tasks to generalize its applicability. On the technical side, we plan to compare purely online adaptation with warm-started or offline-trained controllers, and explore additional opportunities for cost-efficiency tuning. Finally, we are interested in scaling the evaluation across different model families and model sizes to strengthen the external validity of TraceGate.

Data Availability

The replication package of our experiments is available at https://github.com/NicolasSchuler/tracegate/.

References

[1] E. Guimaraes and N. Nascimento, "Ai in the software development lifecycle: Insights and open research questions," in *Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering*, ser. FSE Companion '25. New York, NY, USA: Association for Computing Machinery, 2025, p. 1353–1357. [Online]. Available: https://doi.org/10.1145/3696630.3730538

[2] OpenAI. (2025) Codex. Agentic coding tool by OpenAI. Accessed 2026-03-19. [Online]. Available: https://openai.com/index/introducing-codex/

[3] Anthropic. (2025) Claude code. Agentic coding tool by Anthropic. Accessed 2026-03-19. [Online]. Available: https://docs.anthropic.com/en/docs/agents-and-tools/claude-code/overview

[4] X. Chen, M. Lin, N. Schärli, and D. Zhou, "Teaching large language models to self-debug," in *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net, 2024. [Online]. Available: https://openreview.net/forum?id=KuPixIqPiq

[5] H. Ye, M. Martinez, X. Luo, T. Zhang, and M. Monperrus, "Selfapr: Self-supervised program repair with test execution diagnostics," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '22. New York, NY, USA: Association for Computing Machinery, 2023. [Online]. Available: https://doi.org/10.1145/3551349.3556926

[6] H. Wang, Z. Liu, S. Wang, G. Cui, N. Ding, Z. Liu, and G. Yu, "INTERVENOR: Prompting the coding ability of large language models with the interactive chain of repair," in *Findings of the Association for Computational Linguistics: ACL 2024*, L.-W. Ku, A. Martins, and V. Srikumar, Eds. Bangkok, Thailand: Association for Computational Linguistics, Aug. 2024, pp. 2081–2107. [Online]. Available: https://aclanthology.org/2024.findings-acl.124/

[7] I. Bouzenia, P. T. Devanbu, and M. Pradel, "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, 2025, pp. 2188–2200. [Online]. Available: https://doi.org/10.1109/ICSE55347.2025.00157

[8] J. Huang, W. Ye, W. Sun, J. Zhang, M. Zhang, and Y. Liu, "Tracecoder: A trace-driven multi-agent framework for automated debugging of llm-generated code," *ICSE 2026*, 02 2026.

[9] L. Zhong, Z. Wang, and J. Shang, "Debug like a human: A large language model debugger via verifying runtime execution step by step," in *Findings of the Association for Computational Linguistics, ACL 2024, Bangkok, Thailand and virtual meeting, August 11-16, 2024*, ser. Findings of ACL, L. Ku, A. Martins, and V. Srikumar, Eds., vol. ACL 2024. Association for Computational Linguistics, 2024, pp. 851–870. [Online]. Available: https://doi.org/10.18653/v1/2024.findings-acl.49

[10] S. Kang, B. Chen, S. Yoo, and J. Lou, "Explainable automated debugging via large language model-driven scientific debugging," *Empir. Softw. Eng.*, vol. 30, no. 2, p. 45, 2025. [Online]. Available: https://doi.org/10.1007/s10664-024-10594-x

[11] K. Levin, N. van Kempen, E. D. Berger, and S. N. Freund, "Chatdbg: Augmenting debugging with large language models," *Proc. ACM Softw. Eng.*, vol. 2, no. FSE, pp. 1892–1913, 2025. [Online]. Available: https://doi.org/10.1145/3729355

[12] Y. Wang, Y. Zhang, G. Li, C. Zhi, B. Li, F. Huang, Y. Li, and S. Deng, "Inspectcoder: Dynamic analysis-enabled self repair through interactive llm-debugger collaboration," 2025. [Online]. Available: https://arxiv.org/abs/2510.18327

[13] P. Gao and C. Peng, "More with less: An empirical study of turn-control strategies for efficient coding agents," *CoRR*, vol. abs/2510.16786, 2025. [Online]. Available: https://doi.org/10.48550/arXiv.2510.16786

[14] Y. Du, M. Tian, S. Ronanki, S. Rongali, S. B. Bodapati, A. Galstyan, A. Wells, R. Schwartz, E. A. Huerta, and H. Peng, "Context length alone hurts LLM performance despite perfect retrieval," in *Findings of the Association for Computational Linguistics: EMNLP 2025, Suzhou, China, November 4-9, 2025*, C. Christodoulopoulos, T. Chakraborty, C. Rose, and V. Peng, Eds. Association for Computational Linguistics, 2025, pp. 23 281–23 298. [Online]. Available: https://aclanthology.org/2025.findings-emnlp.1264/

[15] N. F. Liu, K. Lin, J. Hewitt, A. Paranjape, M. Bevilacqua, F. Petroni, and P. Liang, "Lost in the middle: How language models use long contexts," *Trans. Assoc. Comput. Linguistics*, vol. 12, pp. 157–173, 2024. [Online]. Available: https://doi.org/10.1162/tacl_a_00638

[16] S. Lee, Y. Jo, M. Seo, M. Lee, and M. Seo, "Lost in the noise: How reasoning models fail with contextual distractors," *CoRR*, vol. abs/2601.07226, 2026. [Online]. Available: https://doi.org/10.48550/arXiv.2601.07226

[17] P. Zhu, L. Sun, P. S. Yu, and S. Su, "The necessity of a unified framework for llm-based agent evaluation," 2026. [Online]. Available: https://arxiv.org/abs/2602.03238

[18] X. Li, R. Ming, P. Setlur, A. Paladugu, A. Tang, H. Kang, S. Shao, R. Jin, and C. Xiong, "Benchmark test-time scaling of general llm agents," 2026. [Online]. Available: https://arxiv.org/abs/2602.18998

[19] C. Zhang, M. Xia, X. Zhang, D. Madrigal, A. Mallick, S. Kessler, V. Ruehle, and S. Rajmohan, "Budget-aware agentic routing via boundary-guided training," 2026. [Online]. Available: https://arxiv.org/abs/2602.21227

[20] J. Yang, B. Hou, W. Wei, Y. Bao, and S. Chang, "Ares: Adaptive reasoning effort selection for efficient llm agents," 2026. [Online]. Available: https://arxiv.org/abs/2603.07915

[21] C. Le Goues, M. Pradel, A. Roychoudhury, and S. Chandra, "Automatic program repair," *IEEE Software*, vol. 38, no. 4, pp. 22–27, 2021. [Online]. Available: https://ieeexplore.ieee.org/document/9461040/

[22] W. Weimer, S. Forrest, C. Le Goues, and T. Nguyen, "Automatic program repair with evolutionary computation," *Communications of the ACM*, vol. 53, no. 5, pp. 109–116, 2010. [Online]. Available: https://dl.acm.org/doi/10.1145/1735223.1735249

[23] R. Haldar and J. Hockenmaier, "Rating roulette: Self-inconsistency in llm-as-a-judge frameworks," in *Findings of the Association for Computational Linguistics: EMNLP 2025, Suzhou, China, November 4-9, 2025*, C. Christodoulopoulos, T. Chakraborty, C. Rose, and V. Peng, Eds. Association for Computational Linguistics, 2025, pp. 24 986–25 004. [Online]. Available: https://aclanthology.org/2025.findings-emnlp.1361/

[24] A. Paleyes, R. Sendyka, D. Robinson, C. Cabrera, and N. D. Lawrence, "Code roulette: How prompt variability affects llm code generation," in *Proceedings of the 2026 IEEE/ACM 48th International Conference on Software Engineering: Companion Proceedings*, ser. ICSE-Companion '26. New York, NY, USA: Association for Computing Machinery, 2064. [Online]. Available: https://arxiv.org/abs/2506.10204

[25] J. Jiang, F. Wang, J. Shen, S. Kim, and S. Kim, "A survey on large language models for code generation," *ACM Trans. Softw. Eng. Methodol.*, vol. 35, no. 2, Jan. 2026. [Online]. Available: https://doi.org/10.1145/3747588

[26] A. Holtzman, J. Buys, L. Du, M. Forbes, and Y. Choi, "The curious case of neural text degeneration," in *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net, 2020. [Online]. Available: https://openreview.net/forum?id=rygGQyrFvH

[27] J. A. Prenner and R. Robbes, "Out of context: How important is local context in neural program repair?" ser. ICSE '24. New York, NY, USA: Association for Computing Machinery, 2024. [Online]. Available: https://doi.org/10.1145/3597503.3639086

[28] J. Xu, Y. Fu, S. H. Tan, and P. He, "Aligning the objective of llm-based program repair," in *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, 2025, pp. 2548–2560.

[29] Z. Huang, L. Xu, C. Liu, W. Sun, X. Zhang, Y. Lei, M. Yan, and H. Zhang. Dynafix: Iterative automated program repair driven by execution-level dynamic information. [Online]. Available: http://arxiv.org/abs/2512.24635

[30] N. Parasaram, H. Yan, B. Yang, Z. Flahy, A. Qudsi, D. Ziaber, E. T. Barr, and S. Mechtaev, " The Fact Selection Problem in LLM-Based Program Repair ," in *2025 IEEE/ACM 47th ICSE*. Los Alamitos, CA, USA: IEEE Computer Society, May 2025, pp. 2574–2586. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/ICSE55347.2025.00162

[31] R. Ehsani, E. Parra, S. Haiduc, and P. Chatterjee, "Hierarchical knowledge injection for improving llm-based program repair," in *2025 40th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2025, pp. 1440–1452. [Online]. Available: http://arxiv.org/abs/2506.24015

[32] M. Haque, P. Babkin, F. Farmahinifarahani, and M. Veloso, "Towards effectively leveraging execution traces for program repair with code LLMs," in *Proceedings of the 4th International Workshop on Knowledge-Augmented Methods for Natural Language Processing*, W. Shi, W. Yu, A. Asai, M. Jiang, G. Durrett, H. Hajishirzi, and L. Zettlemoyer, Eds. Albuquerque, New Mexico, USA: Association for Computational Linguistics, May 2025, pp. 160–179. [Online]. Available: https://aclanthology.org/2025.knowledgenlp-1.17/

[33] W. Zeng, Y. Huang, and J. He, "Loca-bench: Benchmarking language agents under controllable and extreme context growth," 2026. [Online]. Available: https://arxiv.org/abs/2602.07962

[34] H. Wang, C. Qian, M. Li, J. Qiu, B. Xue, M. Wang, H. Ji, A. Storkey, and K.-F. Wong, "Position: Agent should invoke external tools only when epistemically necessary," 2026. [Online]. Available: https://arxiv.org/abs/2506.00886

[35] C. Qian, E. C. Acikgoz, H. Wang, X. Chen, A. Sil, D. Hakkani-Tür, G. Tur, and H. Ji, "Smart: Self-aware agent for tool overuse mitigation," 2025. [Online]. Available: https://arxiv.org/abs/2502.11435

[36] Y. Yao, S. Huang, E. Dai, Z. Tan, Z. Duan, S. Jia, Y. Jiang, and T. Yang, "ARC: active and reflection-driven context management for long-horizon information seeking agents," *CoRR*, vol. abs/2601.12030, 2026. [Online]. Available: https://doi.org/10.48550/arXiv.2601.12030

[37] Y.-A. Xiao, P. Gao, C. Peng, and Y. Xiong. Reducing cost of llm agents with trajectory reduction. [Online]. Available: http://arxiv.org/abs/2509.23586

[38] P. Panda, R. Magazine, C. Devaguptapu, S. Takemori, and V. Sharma, "Adaptive LLM routing under budget constraints," in *Findings of the Association for Computational Linguistics: EMNLP 2025, Suzhou, China, November 4-9, 2025*, C. Christodoulopoulos, T. Chakraborty, C. Rose, and V. Peng, Eds. Association for Computational Linguistics, 2025, pp. 23 934–23 949. [Online]. Available: https://aclanthology.org/2025.findings-emnlp.1301/

[39] T. Han, Z. Wang, C. Fang, S. Zhao, S. Ma, and Z. Chen. Token-budget-aware llm reasoning. [Online]. Available: http://arxiv.org/abs/2412.18547

[40] J. Wang, S. Jain, D. Zhang, B. Ray, V. Kumar, and B. Athiwaratkun, "Reasoning in token economies: Budget-aware evaluation of llm reasoning strategies," in *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, 2024, pp. 19 916–19 939. [Online]. Available: https://aclanthology.org/2024.emnlp-main.1112

[41] Y. Li, L. Frampton, F. Mora, and E. Polgreen, "Online prompt selection for program synthesis," in *Proceedings of the Thirty-Ninth AAAI Conference on Artificial Intelligence and Thirty-Seventh Conference on Innovative Applications of Artificial Intelligence and Fifteenth Symposium on Educational Advances in Artificial Intelligence*, ser. AAAI'25/IAAI'25/EAAI'25. AAAI Press, 2025. [Online]. Available: https://doi.org/10.1609/aaai.v39i11.33227

[42] C. Shi, K. Yang, Z. Chen, J. Li, J. Yang, and C. Shen, "Efficient prompt optimization through the lens of best arm identification," in *Proceedings of the 38th International Conference on Neural Information Processing Systems*, ser. NIPS '24. Red Hook, NY, USA: Curran Associates Inc., 2024.

[43] H. Tang, K. Hu, J. P. Zhou, S. Zhong, W.-L. Zheng, X. Si, and K. Ellis, "Code repair with llms gives an exploration-exploitation tradeoff," *Advances in Neural Information Processing Systems*, vol. 37, pp. 117 954–117 996, 2024.

[44] J. Langford and T. Zhang, "The epoch-greedy algorithm for contextual multi-armed bandits," in *Proceedings of the 21st International Conference on Neural Information Processing Systems*, ser. NIPS'07. Red Hook, NY, USA: Curran Associates Inc., 2007, p. 817–824.

[45] M. Dudík, J. Langford, and L. Li, "Doubly robust policy evaluation and learning," ser. ICML'11. Madison, WI, USA: Omnipress, 2011, p. 1097–1104.

[46] W. Chu, L. Li, L. Reyzin, and R. Schapire, "Contextual bandits with linear payoff functions," in *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, ser. Proceedings of Machine Learning Research, G. Gordon, D. Dunson, and M. Dudík, Eds., vol. 15. Fort Lauderdale, FL, USA: PMLR, 11–13 Apr 2011, pp. 208–214. [Online]. Available: https://proceedings.mlr.press/v15/chu11a.html

[47] Qwen Team, "Qwen3.5: Towards native multimodal agents," February 2026. [Online]. Available: https://qwen.ai/blog?id=qwen3.5

[48] ggml org, "llama.cpp: Llm inference in c/c++." [Online]. Available: https://github.com/ggml-org/llama.cpp

[49] Y. Wu, Z. Li, J. M. Zhang, and Y. Liu, "Condefects: A complementary dataset to address the data leakage concern for llm-based fault localization and program repair," in *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, ser. FSE 2024. New York, NY, USA: Association for Computing Machinery, 2024, p. 642–646. [Online]. Available: https://doi.org/10.1145/3663529.3663815

[50] R. Tian, Y. Ye, Y. Qin, X. Cong, Y. Lin, Y. Pan, Y. Wu, H. Hui, W. Liu, Z. Liu, and M. Sun, "Debugbench: Evaluating debugging capability of large language models," in *Findings of the Association for Computational Linguistics, ACL 2024, Bangkok, Thailand and virtual meeting, August 11-16, 2024*, ser. Findings of ACL, L. Ku, A. Martins, and V. Srikumar, Eds., vol. ACL 2024. Association for Computational Linguistics, 2024, pp. 4173–4198. [Online]. Available: https://doi.org/10.18653/v1/2024.findings-acl.247

[51] N. Muennighoff, Q. Liu, A. Zebaze, Q. Zheng, B. Hui, T. Y. Zhuo, S. Singh, X. Tang, L. Von Werra, and S. Longpre, "Octopack: Instruction tuning code large language models," in *NeurIPS 2023 workshop on instruction tuning and instruction following*, 2023.

[52] W. G. Cochran, *Sampling Techniques, 3rd Edition*. John Wiley, 1977.

[53] J. Wei, X. Wang, D. Schuurmans, M. Bosma, B. Ichter, F. Xia, E. H. Chi, Q. V. Le, and D. Zhou, "Chain-of-thought prompting elicits reasoning in large language models," in *Proceedings of the 36th International Conference on Neural Information Processing Systems*, ser. NIPS '22. Red Hook, NY, USA: Curran Associates Inc., 2022.

[54] A. Roy, R. Tutunov, X. Ji, M. Zimmer, and H. Bou-Ammar, "The $\mathbf{Y}$-combinator for llms: Solving long-context rot with λ-calculus," 2026. [Online]. Available: https://arxiv.org/abs/2603.20105

[55] X. Hu, F. Niu, J. Chen, X. Zhou, J. Zhang, J. He, X. Xia, and D. Lo, "Assessing and advancing benchmarks for evaluating large language models in software engineering tasks," *ACM Trans. Softw. Eng. Methodol.*, Dec. 2026, just Accepted. [Online]. Available: https://doi.org/10.1145/3786771

[56] Google Deepmind. (2026) Gemma-4. [Online]. Available: https://ai.google.dev/gemma/docs/core/model_card_4

[57] Z.ai. (2026) Glm-4.7-flash. [Online]. Available: https://huggingface.co/unsloth/GLM-4.7-Flash-GGUF