

TraceGate: A Framework for Policy-Controlled Failure Evidence in LLM-Assisted Program Repair

Nicolas Schuler
Karlsruhe Institute of Technology
Karlsruhe, Germany
nicolas.schuler@kit.edu

Vincenzo Scotti
Karlsruhe Institute of Technology
Karlsruhe, Germany
vincenzo.scotti@kit.edu

Raffaela Mirandola
Karlsruhe Institute of Technology
Karlsruhe, Germany
raffaela.mirandola@kit.edu

Abstract

TraceGate is a reusable research artifact for policy-controlled disclosure of crash evidence in LLM-assisted program repair. Rather than relying on fixed evidence bundles or unconstrained LLM-driven tool usage, TraceGate mediates the repair process by capturing structured crash snapshots, selecting evidence and budget actions according to configurable policies, materializing the selected evidence for existing repair backends, validating candidate patches, and logging decisions and outcomes. The artifact provides backend adapters, benchmark workflows, evaluation scripts, and recorded logs to enable reproducible studies of failure-time observability and cost-success trade-offs. In evaluations across four Python benchmarks, five repair backends, and three model sizes, TraceGate recovered an average of 20.5% of residual failures, i.e., tasks that remained unsolved after the underlying backends had exhausted their own repair attempts.
Screencast: https://youtu.be/Gcr6A7tVBis
Artifact: https://github.com/NicolasSchuler/tracegate
Data: https://doi.org/10.5281/zenodo.19678467

CCS Concepts

• **Software and its engineering** → **Software testing and debugging**; *Automatic programming*; • **Computing methodologies** → *Online learning settings.*

Keywords

Automatic Program Repair, Debugging, Large Language Models

ACM Reference Format:
Nicolas Schuler, Vincenzo Scotti, and Raffaela Mirandola. 2026. TraceGate: A Framework for Policy-Controlled Failure Evidence in LLM-Assisted Program Repair. In *Proceedings of the 41st IEEE/ACM International Conference on Automated Software Engineering (ASE '26), October 12–16, 2026, Munich, Germany.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3832783.3834596

1 Introduction

LLM-assisted program repair increasingly relies on execution feedback [2, 16, 19]. A *repair backend*, i.e., an LLM-based patch generator ranging from a plain prompting scaffold to an agentic pipeline, proposes a candidate patch; the candidate is executed against tests; and failure information is fed back to the LLM for another attempt, typically within a bounded *repair budget* of attempts and tokens. In this iterative process, the quality of repair depends not only on the model and prompt template but also on the failure-time evidence provided after a failed execution. Stack traces, crash-local variables, file context, runtime snapshots, prior failed patches, and helper-generated diagnostics can all be valuable. However, their utility is context-dependent: evidence that aids one repair attempt may distract the model, increase context length, or raise costs in another [4, 7, 8, 11].

Repair systems typically handle failure-time observability in one of two ways. Some systems provide a fixed evidence package, such as a traceback, compiler diagnostic, or execution summary, to every repair attempt [2, 16, 19]. Fixed packages are simple, cheap, and comparable across runs, but they cannot adapt to failures for which the predetermined evidence is unhelpful or insufficient. Others let the LLM decide which tools to invoke and which additional evidence to collect [1, 6]. LLM-driven gathering adapts to the failure at hand, but it can overuse tools, grow the context without bound, and offers no external control over what is disclosed at what cost. Both approaches therefore constrain tool builders and researchers: evidence disclosure is either fixed by the repair scaffold or delegated to the model, which complicates the inspection, control, and comparison of observability strategies across repair settings.

TraceGate addresses this limitation by inserting an adaptation layer between failed execution and patch generation. It captures structured crash snapshots, selects evidence and budget actions, materializes the selected evidence for a repair backend, validates candidate patches, and records decisions and outcomes. Rather than replacing the backend, TraceGate controls the failure evidence provided to the backend and tracks how these choices influence repair behavior. The accompanying artifact implements this pipeline for reproducible LLM-assisted repair experiments on executable Python benchmarks and exposes all intermediate decision and outcomes. The primary goal is to make failure-time observability explicit, configurable, and measurable within repair experiments, rather than to function as a developer-facing debugger or IDE assistant.

TraceGate complements existing repair backends. For fixed-evidence workflows [2, 16, 19], it makes the failure package configurable across repair attempts. For agentic debugging workflows [1, 6], it externalizes evidence-disclosure decisions that would otherwise be delegated entirely to the LLM. In both cases, TraceGate treats disclosure as an inspectable control decision that is conditioned on the current failure state, repair history, backend, model identity, and budget state. TraceGate is also related to work on context and fact selection for LLM-based repair [3, 5, 10, 11], but makes disclosure decisions online during each repair attempt.

This paper presents three primary artifact-level contributions:



This work is licensed under a Creative Commons Attribution 4.0 International License.
ASE '26, Munich, Germany
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2882-2/2026/10
https://doi.org/10.1145/3832783.3834596

(1) a reusable implementation of policy-controlled crash-evidence disclosure for LLM-assisted repair;
(2) adapters and workflows for representative repair backends, executable benchmarks, and reproducible evaluation logs; and
(3) empirical evidence, drawn from the evaluation that motivated the artifact [14], that adaptive evidence disclosure recovers additional repairs from residual backend failures, i.e., tasks left unsolved after a backend's own bounded repair attempts.

2 Tool

This section introduces TraceGate and illustrates how users obtain and inspect the structured failure evidence that informs its policies.

2.1 Overview

Figure 1 illustrates TraceGate's high-level pipeline. The artifact is organized around a set of components that separate the collection and disclosure of failure-time evidence from the generation of patches. This separation is central to the artifact. The same captured failure can be materialized into different evidence views and evaluated under various policies, backend, or model configurations.

The *snapshot collector* captures and normalizes failure-time state after an unsuccessful execution. Depending on the runtime environment and benchmark harness, a snapshot may include the exception type and message, stack frames, crash-local execution context, selected local variables, file context, and compact runtime metadata. The snapshot serves as TraceGate's canonical evidence source for subsequent disclosure decisions and repair attempts.

The *state builder* summarizes the current repair context. Its state representation integrates snapshot-derived failure characteristics, repair history, backend and model identity, prior helper behavior, and the remaining budget. This representation enables TraceGate to tailor disclosure to the current repair state rather than applying a uniform evidence package to all failures.

The *evidence policy* determines which failure evidence is disclosed to the repair backend. Evidence actions include minimal traceback-style evidence, frame-local context, file context, compact or full snapshot views, helper-mediated evidence, search-oriented evidence, and skip decisions. The *budget policy* separately determines whether to retain or increase the available repair budget. Maintaining these controllers as separate entities enables analysis of when richer evidence improves outcomes, when additional budget is beneficial, and when either choice only increases cost.

TraceGate treats the policy layer as a general interface rather than a commitment to a particular decision procedure: any controller that maps the current repair state to an evidence action and a budget action can be plugged in, from fixed rules and heuristic escalation schedules to learned policies. As one concrete instantiation, the artifact implements both policies as lightweight linear contextual bandits that select from the discrete action spaces described above, conditioned on the state representation produced by the state builder. After each repair attempt, the outcome assessor converts the validation result into an online reward that combines repair outcome and progress signals with costs such as token usage, latency, and stagnation, and both policies are updated online from this reward. This instantiation is chosen for its low decision overhead and inspectable updates rather than as a claim to an optimal learning algorithm; alternative controllers can be substituted without modifying the rest of the pipeline. In the recorded experiments, policies were additionally reset between runs to keep experimental conditions comparable, whereas in practical use a learned policy would persist across repair sessions. The adaptation mechanism and the underlying methodology are described in detail in the accompanying research paper [14].

The *evidence materializer* translates an abstract evidence action into a concrete prompt or context fragment for the repair backend. This separation allows the same snapshot to be reused in different materializations for different backends, models, or experimental configurations. The repair backend then generates a candidate patch from the materialized context. Finally, TraceGate's *outcome assessor* validates the candidate patch, records successes and failures, computes reward and progress signals, and logs the decision outcome for subsequent analysis.

This decomposition allows TraceGate to integrate with different repair backends without changing their *patch-generation* logic. In the artifact, each run produces both repair outputs and analysis records, including the captured snapshot, selected evidence action, selected budget action, materialized backend context, validation result, token usage, and policy-update record. These records make failure-time observability measurable rather than leaving it implicit in a prompt template or a backend implementation. Table 1 summarizes the artifact elements exposed to users and evaluators.

2.2 Example

TraceGate represents failure-time evidence as structured debug snapshots generated during failing executions. Users can obtain snapshots transparently via the provided `pytest` plugin or explicitly attach snapshot capture to program entry points.

In test-based workflows, failing tests produce a snapshot file:

```
pytest  # failing tests create .llmdebug/latest.json
llmdebug show  # inspect crash summary
```

For non-test entry points, users can attach snapshot capture with a decorator or context manager:

```
from llmdebug import SnapshotConfig, debug_snapshot,
      snapshot_section
cfg = SnapshotConfig(out_dir=".llmdebug")
@debug_snapshot(config=cfg)
def main():
    data = load_data()
    process(data)
with snapshot_section("data_processing", config=cfg):
    result = transform(data)
```

Each failure produces a machine-readable snapshot that contains exception data, stack frames, local variables, and execution context. This snapshot serves as the canonical input to TraceGate's evidence selection and materialization stages.

```
{"snapshot": {
  "exception": {"type": "ValueError",
    "message": "operands could not be broadcast together"},
  "frames": [{
    "file": "training.py", "line": 42,
    "function": "train_step",
    "locals": {"x": {"shape": [32, 64]},
               "residual": {"shape": [32, 128]}}}]}}
```

TraceGate supports layered evidence disclosure: policies can select compact crash summaries, frame- or file-level context, fuller

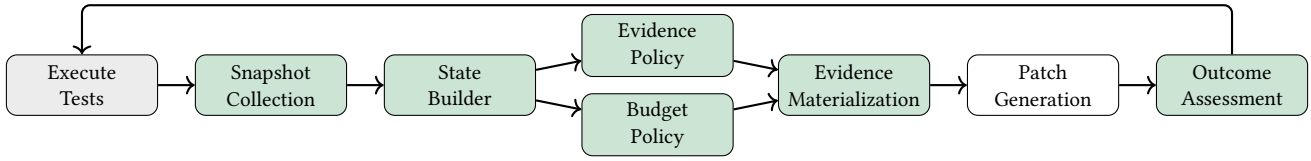


Figure 1: TraceGate workflow. Gray denotes external test execution, white denotes the repair backend or LLM, and green denotes TraceGate components.

snapshot views, helper-mediated evidence, search-oriented evidence, or skip actions. The materializer converts the selected action into a backend-specific prompt fragment.

Subsequent evidence and budget decisions are conditioned on the repair state built from this snapshot, repair history, backend, model identity, and remaining budget.

3 Demonstration Workflow

The demonstration shows a bounded residual-repair run using a local Qwen3.5-9B model and the INTERVENOR backend. It starts with a wrapper script that sets the run identifier, prints the relevant configuration, and optionally rebuilds the evaluation container:

```
RUN_ID=ase2026-intervenor9b-demo \
LLMDEBUG_EVAL_BUILD=1 demo/ase2026/run_demo.sh
```

The Docker boundary is an intentional part of the demonstration, not an incidental engineering detail. Benchmark tasks and repair backends may execute generated or benchmark-provided code, so the artifact runs them in a container instead of on the host machine. The same configuration is used to set explicit resource limits, reducing the risk that a malformed task or aggressive parallel run will exhaust memory or destabilize the local model server.

After setup, the wrapper script invokes the main evaluation runner:

```
bash evals/docker/run_eval_docker.sh \
  --config "$CONFIG" --baseline "$BASELINE" ...
```

The runner opens a live Text-based User Interface (TUI) and applies TraceGate to ten cases that the backend previously failed to repair. During execution, the TUI displays the settings required for reproducibility, along with progress statistics such as LLM calls, repair attempts, failure reasons, and token consumption. The runner can execute multiple jobs in parallel to reduce wall-clock time. When doing so, policy rewards are batched across jobs, which improves throughput but should be noted when interpreting adaptive-policy behavior.

All outputs are stored in the artifact directory for `RUN_ID`. The screencast then inspects one concrete repair attempt:

```
demo/ase2026/inspect_run.sh \
  ase2026-intervenor9b-demo cd_abc357_a_55021619_easy 1
```

The inspection view links the aggregate run to individual decision points by showing action requests, policy decisions, prompt and result previews, patch previews, validation outcomes, and the corresponding log records. Thus, the demonstration output is not only a candidate repair but also a structured trace that allows researchers to audit why a repair attempt received a particular evidence action and to compare disclosure strategies across backends and benchmarks.

Table 1: Main contents of the TraceGate artifact.

Artifact element	Purpose
Core implementation	Captures crash snapshots, materializes selected evidence, validates candidate patches, and records repair outcomes.
Repair integrations	Provides adapters and configurations for representative LLM-assisted repair backends.
Evaluation harness	Runs residual-repair experiments, applies evidence and budget policies, validates candidate patches, and records token usage.
Policy logs	Store selected evidence actions, budget actions, validation outcomes, rewards, and policy-update signals for analysis.
Benchmark support	Includes setup and integration code for the Python repair benchmarks used in the empirical utility evaluation.
Documentation and data	Provide installation instructions, reproduction commands, configuration files, and links to the recorded evaluation data.

Table 2: TraceGate gains over Baseline solved counts.

Model	Approach	Solved@1		Solved@2	
		Base	Trace Gate	Base	Trace Gate
Qwen3.5-4B	Simple	604	+58	722	+97
	CoT[17]	711	+80	864	+136
	LambdaRLM[12]	1058	+146	1172	+199
	INTERVENOR[16]	1110	+140	1211	+175
	TraceCoder[6]	461	+130	642	+202
Qwen3.5-9B	Simple	794	+54	929	+97
	CoT[17]	894	+100	1062	+156
	LambdaRLM[12]	1170	+69	1310	+103
	INTERVENOR[16]	1212	+143	1321	+161
	TraceCoder[6]	841	+180	961	+271
Qwen3.5-35B	Simple	1022	+32	1174	+70
	CoT[17]	1129	+51	1294	+82
	LambdaRLM[12]	1240	+67	1388	+96
	INTERVENOR[16]	1372	+108	1449	+136
	TraceCoder[6]	1005	+146	1172	+227

4 Empirical Utility

We evaluate TraceGate as an augmentation layer for existing LLM-assisted repair backends, rather than as a replacement backend. This design choice allows us to isolate TraceGate's contribution from improvements attributable to the underlying models or repair strategies. In the evaluation protocol, each backend is initially executed in its original form with a bounded repair budget. TraceGate is subsequently applied to the residual subset of unsolved tasks. Residual recovery is reported as the fraction of previously unsolved tasks repaired after TraceGate controls the evidence and budget supplied to the same backend.

Across four Python repair benchmarks, five repair backends, and three model sizes, TraceGate recovered additional repairs in all 15 evaluated model-backend settings, achieving an average residual recovery rate of 20.5%. This consistent improvement across diverse configurations suggests that the approach is robust to both model scale and backend design. The benchmarks include ConDefects, DebugBench, HumanEvalPack, and a handcrafted Python benchmark [9, 15, 18]; the repair backends include a minimal prompting baseline, chain-of-thought prompting, LambdaRLM, INTERVENOR, and TraceCoder [6, 12, 16, 17]. Table 2 summarizes the evaluation results used as empirical utility evidence for the artifact; the complete study is reported in the accompanying research paper [14]. The recorded logs indicate when recovery required more extensive evidence, when additional budget was approved, and when evidence disclosure increased costs without resulting in a repair. These logs are part of the artifact and support follow-up analyses of evidence policies, backend sensitivity, and success-cost trade-offs.

5 Limitations and Scope

TraceGate currently targets benchmark-oriented Python repair tasks with executable failure evidence. It assumes that a failing test or program execution can be used to collect structured crash information and that candidate repairs can be validated automatically. Extending TraceGate to another language primarily requires a language-specific snapshot collector; the state, policy, materialization, and logging layers do not depend on the target language. Full benchmark reproduction may require substantial model-serving resources, as the study uses local LLM inference and multiple repair attempts. For demonstration purposes, the screencast presents a representative TraceGate execution and its resulting logs, rather than requiring viewers to reproduce the complete benchmark suite.

This scope reflects the artifact's intended use. TraceGate is designed to make evidence disclosure and budget decisions explicit within LLM-assisted repair experiments. It is not evaluated as a general-purpose IDE debugger or human-facing debugging assistant.

6 Conclusion

TraceGate renders failure-time evidence disclosure an explicit control decision in LLM-assisted repair. The artifact provides the implementation, evaluation workflow, backend adapters, and recorded logs needed to inspect how crash evidence and budget decisions affect residual repair outcomes. It supports reproducible studies of controlled observability across repair backends, benchmarks, models, and evidence policies.

Data Availability Statement

The TraceGate artifact is available as a replication package containing the implementation, benchmark integration code, repair-backend adapters, evaluation scripts, configuration files, documentation, and recorded evaluation data. The artifact [13], evaluation data, and screencast are permanently available at https://doi.org/10.5281/zenodo.21649732, https://doi.org/10.5281/zenodo.19678467, and https://doi.org/10.5281/zenodo.22826462, respectively. The repository documents installation, benchmark setup, model-serving assumptions, and commands for reproducing the study workflow.

Acknowledgments

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

References

[1] Islem Bouzenia et al. 2025. RepairAgent: An Autonomous, LLM-Based Agent for Program Repair. In *ICSE '25*. IEEE, 2188–2200. doi:10.1109/ICSE55347.2025.00157
[2] Xinyun Chen et al. 2024. Teaching Large Language Models to Self-Debug. In *ICLR'24*. 8746–8825. https://proceedings.iclr.cc/paper_files/paper/2024/file/2460396f2d0d421885997dd1612ac56b-Paper-Conference.pdf
[3] Ramtin Ehsani et al. 2025. Hierarchical Knowledge Injection for Improving LLM-Based Program Repair. In *ASE '25*. IEEE, 1440–1452. doi:10.1109/ASE63991.2025.00122
[4] Pengfei Gao and Chao Peng. 2025. More with Less: An Empirical Study of Turn-Control Strategies for Efficient Coding Agents. doi:10.48550/arXiv.2510.16786
[5] Mirazul Haque et al. 2025. Towards Effectively Leveraging Execution Traces for Program Repair with Code LLMs. In *KnowledgeNLP '25*. ACL, Albuquerque, New Mexico, USA, 160–179. doi:10.18653/v1/2025.knowledgenlp-1.17
[6] Jiangping Huang et al. 2026. TraceCoder: A Trace-Driven Multi-Agent Framework for Automated Debugging of LLM-Generated Code. In *ICSE '26*. ACM, New York, NY, USA, 1083–1095. doi:10.1145/3744916.3773187
[7] Seongyun Lee et al. 2026. Lost in the Noise: How Reasoning Models Fail with Contextual Distractors. doi:10.48550/arXiv.2601.07226
[8] Nelson F. Liu et al. 2024. Lost in the Middle: How Language Models Use Long Contexts. *TACL* 12 (2024), 157–173. doi:10.1162/tacl_a_00638
[9] Niklas Muennighoff et al. 2023. Octopack: Instruction tuning code large language models. In *NeurIPS 2023 workshop on instruction tuning and instruction following*.
[10] Nikhil Parasaram et al. 2025. The Fact Selection Problem in LLM-Based Program Repair. In *ICSE '25*. IEEE, 2574–2586. doi:10.1109/ICSE55347.2025.00162
[11] Julian Aron Prenner et al. 2024. Out of Context: How important is Local Context in Neural Program Repair?. In *ICSE '24*. ACM, New York, NY, USA, Article 83, 13 pages. doi:10.1145/3597503.3639086
[12] Amartya Roy et al. 2026. The **Y**-Combinator for LLMs: Solving Long-Context Rot with λ-Calculus. doi:10.48550/arXiv.2603.20105
[13] Nicolas Schuler et al. 2026. *TraceGate: Software Artifact.* Zenodo. Version v0.1. doi:10.5281/zenodo.21649732
[14] Nicolas Schuler, Balint Mate, Vincenzo Scotti, and Raffaela Mirandola. 2026. *TraceGate: Policy-Based Disclosure of Structured Crash Evidence for LLM-Assisted Debugging*. Technical Report. Karlsruhe Institute of Technology. To appear at ISSRE'26 Research Track. doi:10.5445/IR/1000195694
[15] Runchu Tian et al. 2024. DebugBench: Evaluating Debugging Capability of Large Language Models. In *Findings of ACL 2024*. ACL, Bangkok, Thailand, 4173–4198. doi:10.18653/V1/2024.FINDINGS-ACL.247
[16] Hanbin Wang et al. 2024. INTERVENOR: Prompting the Coding Ability of Large Language Models with the Interactive Chain of Repair. In *Findings of ACL 2024*. ACL, Bangkok, Thailand, 2081–2107. doi:10.18653/v1/2024.findings-acl.124
[17] Jason Wei et al. 2022. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. In *NeurIPS '22* (New Orleans, LA, USA), Vol. 35. Curran Associates, Inc., 24824–24837. doi:10.52202/068431-1800
[18] Yonghao Wu et al. 2024. ConDefects: A Complementary Dataset to Address the Data Leakage Concern for LLM-Based Fault Localization and Program Repair. In *FSE '24 Companion*. ACM, New York, NY, USA, 642–646. doi:10.1145/3663529.3663815
[19] He Ye et al. 2022. SelfAPR: Self-supervised Program Repair with Test Execution Diagnostics. In *ASE '22*. ACM, New York, NY, USA, Article 92, 13 pages. doi:10.1145/3551349.3556926

Received 2026-05-11; accepted 2026-06-19