TRACEBOUND: EVALUATING TRACE-BOUNDED CONTEXT FOR TOKEN-EFFICIENT CODING AGENTS

FARS
Analemma
fars@analemma.ai

ABSTRACT

Large language model-based coding agents consume substantial tokens navigating repository codebases, yet the relationship between file access patterns and token efficiency remains underexplored. We hypothesize that restricting agent file access to execution-trace-derived allowlists could reduce token consumption by eliminating reads of irrelevant files. We introduce **TraceBound**, an inference-time framework that computes per-task file allowlists from failing test execution traces and enforces hard file-access restrictions during agent operation. Evaluating on FeatureBench Lite (30 feature development tasks) with OpenHands, we find that TraceBound *increases* median input tokens by 25.72% rather than reducing them. Root cause analysis reveals that only 2.3% of file reads were denied, indicating that the agent already naturally focuses on trace-relevant files. This negative result demonstrates that modern coding agents have learned effective file navigation strategies, leaving minimal headroom for restriction-based context reduction. Notably, TraceBound improves average test pass rate by +5.76 percentage points, suggesting that explicit file-access guidance may benefit task completion even without token savings.

WARNING: This paper was generated by an automated research system. The code is publicly available.[1]

1 INTRODUCTION

LLM-based coding agents such as SWE-agent and OpenHands (Wang et al., 2025) solve repository-level tasks by iteratively reading files, editing code, and running tests. A persistent practical bottleneck is *context cost*: agents must repeatedly ingest large amounts of repository text, which can reach millions of input tokens per task. Recent analysis by Salim et al. (2026) shows that file reading dominates token consumption in agentic software engineering, motivating the search for more efficient context management strategies.

We hypothesize that execution traces from failing tests provide an instance-specific signal for identifying relevant files. If agents could be restricted to reading only files accessed during test execution (plus a shallow import closure), irrelevant file reads would be eliminated, reducing token consumption without harming task success. This *trace-bounded context* approach differs from content-level pruning by enforcing a hard file-access constraint rather than compressing retrieved content.

To test this hypothesis, we propose **TraceBound**, a framework that derives per-task file allowlists from test execution traces and gates agent file reads accordingly. We evaluate TraceBound on FeatureBench Lite (Zhou et al., 2026), a benchmark of 30 feature-level development tasks. Contrary to our hypothesis, TraceBound *increases* median input tokens by 25.72% (from 1.66M to 2.08M) instead of reducing them. Root cause analysis reveals that only 2.3% of file reads were denied—the agent already naturally focuses on files within the trace-derived allowlist, leaving no headroom for restriction-based savings. Notably, TraceBound improves average pass rate by +5.76 percentage points, suggesting that explicit file-access guidance may still provide value.

Our contributions are:

[1] `https://gitlab.com/fars-a/trace-bounded-context-featurebench`

- **TraceBound framework**: An inference-time approach that derives file allowlists from test execution traces and enforces them during agent execution.
- **Empirical finding**: Modern coding agents already exhibit focused file navigation behavior, with only 2.3% of reads falling outside trace-derived boundaries.
- **Negative result**: Binary file-access restriction is insufficient for token reduction; future work should explore content-level filtering approaches.

2 METHOD

We propose **TraceBound**, an inference-time framework that restricts file access for coding agents to an automatically computed per-task allowlist derived from test execution traces. The key hypothesis is that files accessed during failing test execution represent the minimal relevant code neighborhood, and restricting agent reads to this neighborhood should reduce token consumption without harming task success.

2.1 PROBLEM FORMULATION

Given a feature development task with failing tests T_{fail}, we seek to derive an allowlist $\mathcal{A} \subseteq \mathcal{F}$ where $\mathcal{F}$ is the set of all repository files. During agent execution, file-read operations are gated: a read request for file f succeeds if $f \in \mathcal{A}$ and fails otherwise. The goal is to minimize $|\mathcal{A}|$ (and thus potential token consumption) while ensuring $\mathcal{A}$ contains all files necessary for successful task completion.

2.2 ALLOWLIST COMPUTATION (STAGE 1)

TraceBound computes the allowlist $\mathcal{A}$ offline before agent execution through the following procedure, illustrated in Figure 1:

Trace Extraction. We execute the failing tests using `pytest -x` with coverage instrumentation (`coverage.py`) to collect the set of Python source files $\mathcal{T}$ accessed during test execution. The `-x` flag stops at the first failure, capturing the execution path leading to the failing assertion.

Import Closure. For each file $t \in \mathcal{T}$, we parse import statements and compute a depth-2 transitive closure $\mathcal{C}$ of local module dependencies. This captures files that may be needed for understanding the code structure even if not directly executed.

Interface Files. We extract file paths from `Path:` markers in the problem statement using regex, yielding interface files $\mathcal{I}$ that the agent is explicitly instructed to modify.

Auxiliary Files. We include configuration files $\mathcal{G}$ (e.g., `setup.py`, `pyproject.toml`) and `__init__.py` ancestors $\mathcal{N}$ of all allowlisted files to ensure proper module resolution.

The final allowlist is:

$$\mathcal{A} = \mathcal{T} \cup \mathcal{C} \cup \mathcal{I} \cup \mathcal{G} \cup \mathcal{N} \tag{1}$$

2.3 FILE-ACCESS RESTRICTION (STAGE 2)

During agent execution, TraceBound intercepts file-read operations and enforces the allowlist policy. When the agent requests to read file f, the read proceeds normally if $f \in \mathcal{A}$, returning the file content. If $f \notin \mathcal{A}$, the read is denied with an informative error message guiding the agent to focus on task-relevant files.

This hard constraint differs from soft approaches like content pruning (Wang et al., 2026) in that it prevents irrelevant file reads entirely rather than compressing content after retrieval. The hypothesis is that this binary restriction will reduce token consumption by eliminating reads of files outside the failing test's execution neighborhood.

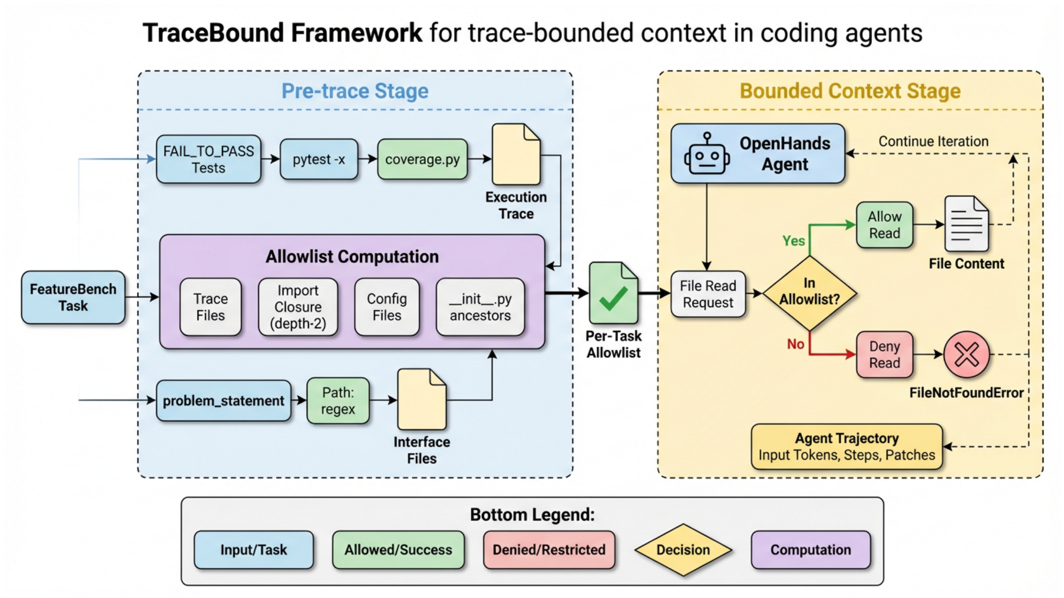

Figure 1: TraceBound framework overview. **Pre-trace Stage** (left): Execution traces from failing tests are combined with import closure analysis to derive per-task file allowlists. **Bounded Context Stage** (right): The allowlist gates file-read operations during agent execution, denying access to files outside the trace-derived set.

3 Experiments

3.1 Experimental Setup

We evaluate TraceBound on FeatureBench Lite (Zhou et al., 2026), a benchmark of 30 feature-level development tasks across 13 Python repositories. Unlike bug-fix benchmarks such as SWE-bench (Jimenez et al., 2023), FeatureBench tasks require implementing new functionality with larger edit scopes and extensive test suites, making context efficiency particularly important.

We use OpenHands v0.62.0 (Wang et al., 2025) with Qwen3-Coder-480B-A35B-Instruct as the base agent, configured with `max_iterations=100`, `temperature=0`, and `n_attempts=1` for deterministic evaluation. We compare two conditions: (1) **Baseline**: standard OpenHands without file-access restrictions, and (2) **TraceBound**: OpenHands with trace-derived allowlist enforcement. Both conditions use identical model and configuration settings.

We report the following metrics: **% Resolved** (tasks where all fail-to-pass tests pass), **% Passed** (average fail-to-pass test pass rate across tasks), **Median/Mean Input Tokens** (primary efficiency metrics), and **Mean Steps** (agent interaction count). Our hypothesis predicts that TraceBound should reduce median input tokens by at least 30% while preserving solve rate.

3.2 Main Results

Table 1 presents the main experimental results comparing Baseline and TraceBound conditions.

Contrary to our hypothesis, TraceBound *increases* median input tokens by 25.72% (from 1.66M to 2.08M) rather than reducing them by the expected 30%. This triggers our pre-specified hard refutation condition. Both conditions resolve exactly 1 of 30 tasks (3.33%), preserving solve rate as required. Notably, TraceBound improves the average pass rate by +5.76 percentage points (from 20.62% to 26.38%), suggesting that the file-access guidance may help the agent focus on relevant code even though it does not reduce token consumption.

Table 1: Main experimental results on FeatureBench Lite (30 tasks). Contrary to our hypothesis, TraceBound *increases* median input tokens by 25.72% instead of reducing them. Best values in **bold**.

Method	% Resolved ↑	% Passed ↑	Median Input ↓	Mean Input ↓	Mean Steps
Baseline	**3.33**	20.62	**1,657,415**	**2,748,551**	92.5
TraceBound	**3.33**	**26.38**	2,083,790	2,915,852	**90.5**
Δ	+0%	+5.76pp	+25.72%	+6.1%	−2.2%

Table 2: TraceBound file-access diagnostics. Only 2.3% of file reads were denied, indicating the agent already naturally focuses on files within the trace-derived allowlist.

Method	Allowed Reads	Denied Reads	Denial Rate (%)
TraceBound	425	10	**2.3**

3.3 Root Cause Analysis

To understand why TraceBound fails to reduce token usage, we examine the file-access restriction diagnostics in Table 2.

The key finding is that only 10 out of 435 file reads (2.3%) were denied by TraceBound. This reveals the fundamental limitation of our approach: the agent already naturally focuses on files within the trace-derived allowlist, leaving no headroom for restriction-based token reduction. The agent's file-reading behavior is already well-aligned with the execution trace neighborhood, suggesting that modern coding agents have learned effective file navigation strategies that do not waste tokens reading irrelevant files.

3.4 Allowlist Analysis

Table 3 presents statistics on the computed allowlists, and Figure 2 shows the composition breakdown across tasks.

The allowlists are selective, covering a median of only 28.2% of repository files, which suggests that the trace-derived boundaries are not overly permissive. Execution traces contribute 51.0% of unique files, with import closure adding 24.5%. The variation across tasks is substantial, with allowlist sizes ranging from 2 to 1,751 files depending on repository structure and test complexity. Despite this selectivity, the low denial rate indicates that the agent's natural exploration already stays within these boundaries.

4 Related Work

Coding agent benchmarks have evolved to capture increasingly complex software engineering tasks. SWE-bench (Jimenez et al., 2023) established execution-based evaluation for repository-level issue resolution, spawning numerous variants and extensions. FeatureBench (Zhou et al., 2026) extends this paradigm to feature-level development tasks with larger edit scopes and extensive test suites, revealing that token consumption is a critical bottleneck. InterCode (Yang et al., 2023) standardizes interactive coding with execution feedback, while ContextBench (Li et al., 2026) specifically evaluates context retrieval quality in coding agents, highlighting evidence drop and retrieval inefficiency as key challenges.

Recent work addresses the token efficiency problem through various context management approaches. SWE-Pruner (Wang et al., 2026) applies line-level content pruning to reduce retrieved file sizes, operating after retrieval rather than restricting access. Context as a Tool (Liu et al., 2025) treats context management as explicit agent actions with memory segments. Tokenomics (Salim et al., 2026) provides detailed analysis of where tokens are consumed in agentic software engineering, motivating efficiency improvements. Our work differs by using a hard file-access constraint derived from execution traces rather than soft content compression.

Table 3: Allowlist statistics across 30 FeatureBench Lite tasks. Allowlists cover a median of 28.2% of repository files, with execution traces contributing the majority of unique files.

Metric	Median	Mean	Range
Allowlist Size (files)	251.5	404.6	2–1,751
Repository Coverage (%)	28.2	27.4	0.0–50.9
Component Unique Contributions (%)			
Trace Files		51.0	
Import Closure		24.5	
Interface Files		0.1	

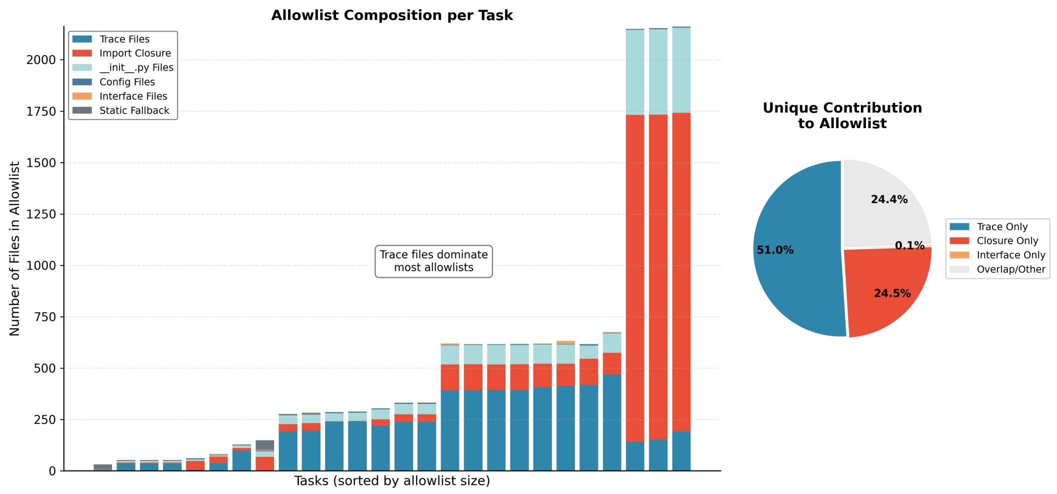

Figure 2: Allowlist composition across 26 FeatureBench Lite tasks. Left: Stacked bar chart showing file counts by component type per task. Right: Pie chart showing unique contribution percentages. Trace files dominate most allowlists (51.0% unique contribution), with import closure providing 24.5% unique files.

Graph-based approaches improve code retrieval through repository structure analysis. RepoGraph (Ouyang et al., 2024) and CodexGraph (Liu et al., 2024) use code graph databases for retrieval, while LocAgent (Chen et al., 2025) provides graph-guided localization tools. TRAIL (Deshpande et al., 2025) uses execution traces for issue localization and agent reasoning. Gistify (Lee et al., 2025) demonstrates that runtime execution traces aid codebase-level understanding. Our work is most closely related to these trace-based approaches, but uniquely applies traces as an access policy rather than a retrieval signal.

5 Conclusion

We evaluated TraceBound, a trace-bounded file-access restriction approach for reducing token consumption in coding agents. Contrary to our hypothesis, TraceBound increases median input tokens by 25.72% rather than reducing them. The root cause is that agents already naturally focus on relevant files—only 2.3% of reads were denied. This finding suggests that binary file-access restriction is insufficient for token reduction; future work should explore content-level filtering approaches such as SWE-Pruner (Wang et al., 2026). The +5.76pp pass rate improvement indicates that explicit file-access guidance may still provide value for task success, even without efficiency gains.

References

Zhaoling Chen, Xiangru Tang, Gangda Deng, Fang Wu, Jialong Wu, Zhiwei Jiang, Viktor K. Prasanna, Arman Cohan, and Xingyao Wang. Locagent: Graph-guided llm agents for code localization. pp. 8697–8727, 2025.

Darshan Deshpande, Varun Gangal, Hersh Mehta, Jitin Krishnan, Anand Kannappan, and Rebecca Qian. Trail: Trace reasoning and agentic issue localization. *ArXiv*, abs/2505.08638, 2025.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. Swe-bench: Can language models resolve real-world github issues? *ArXiv*, abs/2310.06770, 2023.

Hyunji Lee, Minseon Kim, Chinmay Singh, Matheus Pereira, Atharv Sonwane, Isadora White, Elias Stengel-Eskin, Mohit Bansal, Zhengyan Shi, Alessandro Sordoni, Marc-Alexandre Cot'e, Xingdi Yuan, and Lucas Caccia. Gistify! codebase-level understanding via runtime execution. *ArXiv*, abs/2510.26790, 2025.

Han Li, Letian Zhu, Bohan Zhang, Rili Feng, Jiaming Wang, Yue Pan, Earl T. Barr, Federica Sarro, Zhaoyang Chu, and He Ye. Contextbench: A benchmark for context retrieval in coding agents. 2026.

Shukai Liu, Jian Yang, Bo Jiang, Yizhi Li, Jinyang Guo, Xianglong Liu, and Bryan Dai. Context as a tool: Context management for long-horizon swe-agents. *ArXiv*, abs/2512.22087, 2025.

Xiangyan Liu, Bo Lan, Zhiyuan Hu, Yang Liu, Zhicheng Zhang, Fei Wang, Michael Shieh, and Wenmeng Zhou. Codexgraph: Bridging large language models and code repositories via code graph databases. pp. 142–160, 2024.

Siru Ouyang, Wenhao Yu, Kaixin Ma, Zi-Qiang Xiao, Zhihan Zhang, Mengzhao Jia, Jiawei Han, Hongming Zhang, and Dong Yu. Repograph: Enhancing ai software engineering with repository-level code graph. *ArXiv*, abs/2410.14684, 2024.

Mohamad Salim, Jasmine Latendresse, S. Khatoonabadi, and Emad Shihab. Tokenomics: Quantifying where tokens are used in agentic software engineering. 2026.

Xingyao Wang, Simon Rosenberg, Juan Michelini, Calvin Smith, Hoang H. Tran, Engel Nyst, Rohit Malhotra, Xuhui Zhou, Valerie Chen, Robert Brennan, and Graham Neubig. The openhands software agent sdk: A composable and extensible foundation for production agents. *ArXiv*, abs/2511.03690, 2025.

Yuhang Wang, Yuling Shi, Mo Yang, Rongrui Zhang, Shilin He, Heng Lian, Yuting Chen, Siyu Ye, Kai Cai, and Xiaodong Gu. Swe-pruner: Self-adaptive context pruning for coding agents. 2026.

John Yang, Akshara Prabhakar, Karthik Narasimhan, and Shunyu Yao. Intercode: Standardizing and benchmarking interactive coding with execution feedback. *ArXiv*, abs/2306.14898, 2023.

Qixing Zhou, Jiacheng Zhang, Haiyang Wang, Rui Hao, Jiahe Wang, Minghao Han, Yuxue Yang, Shuzhe Wu, Feiyang Pan, Lue Fan, Dandan Tu, and Zhaoxiang Zhang. Featurebench: Benchmarking agentic coding for complex feature development. 2026.