CONTEXTBENCH: A BENCHMARK FOR CONTEXT RETRIEVAL IN CODING AGENTS

Anonymous authors
Paper under double-blind review

ABSTRACT

LLM-based coding agents have shown strong performance on automated issue resolution benchmarks, yet existing evaluations largely focus on final task success, providing limited insight into how agents retrieve and use code context during problem solving. We introduce CONTEXTBENCH, a process-oriented evaluation of context retrieval in coding agents. CONTEXTBENCH consists of *1,136* issue-resolution tasks from 66 repositories across eight programming languages, each augmented with human-annotated gold contexts. CONTEXTBENCH implements an automated evaluation framework that tracks agent trajectories and measures context recall, precision, and efficiency throughout issue resolution. Comprehensive experiments on CONTEXTBENCH across seven frontier LLMs and seven coding agents reveal that simple shell-based agents achieve the strongest overall context retrieval, while specialized retrieval mechanisms provide no consistent gains, echoing the "Bitter Lesson" for coding-agent design. Most evaluated LLMs show higher recall than precision at the block and line levels, and substantial gaps exist between explored and utilized context. CONTEXTBENCH augments existing end-to-end benchmarks with intermediate gold-context metrics that unbox the issue-resolution process. These contexts offer valuable intermediate signals for guiding LLM reasoning in software tasks.

1 INTRODUCTION

Large Language Model (LLM) coding agents, exemplified by SWE-agent (Yang et al., 2024), OpenHands (Wang et al., 2025), and Agentless (Xia et al., 2024) have recently demonstrated strong proficiency in real-world software engineering tasks, including bug fixing (Jimenez et al., 2024; Pan et al., 2025), feature implementation (Rashid et al., 2025; Deng et al., 2025a), and interaction with terminal-based execution environments (Merrill et al., 2026; Chu et al., 2026). These agents are autonomous systems capable of reasoning and tool use, enabling them to solve complex, repository-scale tasks in real-world development scenarios.

Limitations of Existing Benchmarks. Advancements in coding agents are typically assessed on benchmarks such as SWE-bench (Jimenez et al., 2024) and SWE-Bench Pro (Deng et al., 2025b), which curate tasks from real-world GitHub issues and require agents to submit patches validated by test suites. However, these benchmarks prioritize end-to-end task success rates (*e.g.,* Pass@k) and ignore *how coding agents arrive at their solutions*. In particular, they overlook the evaluation of context retrieved from large codebases, which underpins the agent's reasoning. Final task success alone does not reveal whether an agent retrieved the relevant code efficiently, inspected excessive irrelevant context, or failed to retain evidence it had already found. Measuring intermediate contexts makes these behaviors observable alongside patch correctness. We therefore argue for a context-based evaluation system that opens the black box of software tasks and addresses the key question: *"How do LLM agents retrieve and use critical code context when resolving software engineering tasks?"*

CONTEXTBENCH: Beyond End-to-End Benchmarking. To address this question, we develop a semi-automated pipeline for curating *Gold Contexts*, defined as standard sets of repository artifacts that expert developers identify as necessary to resolve a given issue. We begin by sampling high-quality tasks from four widely used benchmarks, *i.e.*, SWE-bench Verified (Jimenez et al., 2024), Multi-SWE-bench (Zan et al., 2025), SWE-PolyBench PB500 (Rashid et al., 2025), and SWE-Bench Pro (Deng et al., 2025b). After deduplication via rule-based matching and embedding-based similarity detection, we select challenging tasks based on three difficulty metrics, *i.e.*, agent solvability; edit

Code and data are available at https://anonymous.4open.science/r/ContextBench-637B.

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

scope; and edit dispersion based on gold patches. For each selected task, human experts iteratively annotate and refine the required context in a human-in-the-loop process, ensuring compact and sufficient gold contexts for issue resolution.

Building on this pipeline, we introduce CONTEXTBENCH, an extensive benchmark designed to evaluate the context retrieval ability of coding agents in software engineering tasks, enabling fine-grained analysis of agent behavior beyond conventional end-to-end issue resolution metrics. CONTEXTBENCH comprises *1,136* issue resolution tasks from *66* code repositories across *8* programming languages. These tasks are annotated with *522,115* lines of human-verified gold contexts covering *23,116* classes and functions within *4,548* files. To facilitate efficient evaluation, we also identify a Lite subset of *500* tasks based on task difficulties.

We further introduce an automated evaluation framework with a systematic suite of metrics to assess how effectively LLM agents retrieve code contexts during problem solving. These metrics evaluate code context recall and precision at different stages of task resolving, complementing the final success rate. Specifically, we instrument the agent's trajectory to record all code regions it inspects. We then utilize the `tree-sitter`[1] tool to parse the corresponding repository and map both gold and agent contexts onto a shared coordinate system that captures their file paths, AST blocks, and line ranges. Based on this alignment, we compute recall, precision, and F1 using interval overlap at three granularities: file, block, and line levels. Radar plots of overall and per-language retrieval performance are provided in Appendix K.

Take-Aways. We benchmark seven state-of-the-art LLMs and seven widely used coding agents. Our analysis reveals several key findings: ❶ **Simple shell-based retrieval is sufficient; specialized retrieval is over-engineered.** The three shell-based agents occupy the top three positions in F1 at both the file and block levels, while Claude Code and Codex also achieve the two highest Pass@1 scores. Retrieval mechanisms using embeddings, specialized interfaces, or graphs provide no consistent improvement, echoing the "Bitter Lesson": general model reasoning paired with flexible, low-level tools can outperform hand-designed retrieval machinery. ❷ **CONTEXTBENCH is challenging for state-of-the-art LLMs.** State-of-the-art LLMs struggle to retrieve effective code contexts, often covering relevant information while introducing substantial noise that undermines retrieval precision. ❸ **Most LLMs favor recall over precision at finer granularities.** Most evaluated LLMs achieve higher recall than precision at the block and line levels, retrieving substantial context outside the gold reference and yielding limited F1 scores. ❹ **Balanced retrieval improves performance.** Models that balance retrieval frequency and context granularity achieve higher Pass@1, while aggressive retrieval mainly increases token consumption. ❺ **Significant gaps exist between retrieved and utilized context.** Agents often inspect gold-relevant code but fail to retain or use it in final patch generation, highlighting consolidation as a key bottleneck.

Contributions. To summarize, our work provides three key contributions. **(i)** We construct CONTEXTBENCH, containing 1,136 full instances for evaluating context retrieval in coding agents, each of which provides expert-annotated, verified gold context for assessing intermediate states of issue resolution. **(ii)** CONTEXTBENCH moves beyond final success-rate evaluation by enabling dynamic, process-oriented assessment across issue resolution stages through tracking and comparing agent-retrieved and gold contexts, providing an important intermediate signal for rewarding agent behavior. **(iii)** We evaluate seven frontier LLMs and seven coding agents on context retrieval recall and precision, yielding new insights into how LLMs and coding agents retrieve code context during problem solving.

2 CONTEXTBENCH: A BENCHMARK TO ASSESS CONTEXT RETRIEVAL ABILITY

As illustrated in Figure 1, we present CONTEXTBENCH, a benchmark for evaluating the agent's context retrieval ability in software engineering tasks, which is curated through three key steps: **(1) Task Deduplication.** We pool tasks from multiple issue resolution benchmarks and eliminate exact and near-duplicate tasks through both rule-based and embedding-based detection. **(2) Task Selection.** We select challenging tasks using three difficulty metrics, *i.e.*, agent solvability, the scope and dispersion of edits in the ground-truth patch, prioritizing tasks that are complex or remain unsolved by existing agents. **(3) Expert Annotation.** For each selected task, expert developers iteratively annotate gold contexts in a human-in-the-loop process by tracing essential code dependencies from

[1] https://tree-sitter.github.io/tree-sitter/

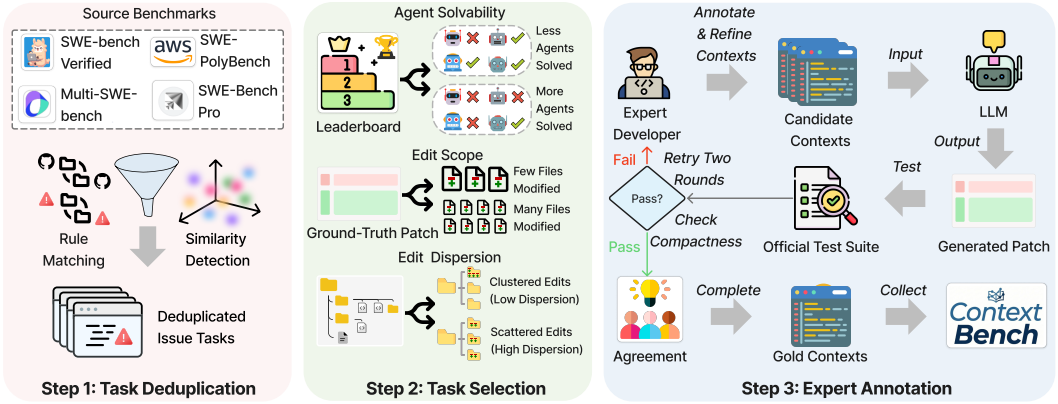


Figure 1: **An overview of the CONTEXTBENCH construction pipeline.** CONTEXTBENCH is curated through three key steps: **(1) Task Deduplication** removes exact and near-duplicate tasks from multiple issue resolution benchmarks using rule-based and embedding-based detection. **(2) Task Selection** identifies challenging tasks based on agent solvability and the scope and dispersion of edits in ground-truth patches. **(3) Expert Annotation** employs a human-in-the-loop procedure in which expert developers trace code dependencies to construct gold contexts, which are validated through LLM-based patch generation and inter-annotator agreement checks.

the ground-truth patches. We then validate each annotated context by evaluating whether an LLM (*e.g.*, GPT-5), conditioned solely on the context, can generate a patch that passes the official test suite or whether the context achieves high inter-annotation consistency. Based on this pipeline, we collect 1,136 task instances with verified gold contexts across 66 repositories and 8 programming languages.

2.1 STEP 1: TASK DEDUPLICATION

We curate CONTEXTBENCH from four widely used issue resolution benchmarks, *i.e.*, SWE-bench Verified (Jimenez et al., 2024) (500 tasks), Multi-SWE-bench (Zan et al., 2025) (1632 tasks), SWE-PolyBench PB500 (Rashid et al., 2025) (500 tasks), and SWE-Bench Pro (Deng et al., 2025b) (1865 tasks), totaling 4,497 tasks across 8 programming languages. To ensure uniqueness, we remove exact duplicates through metadata matching and filter near-duplicates using embedding-based semantic similarity, followed by manual verification of borderline cases. This process results in 3,100 unique real-world issue resolution tasks. More details are provided in Appendix A.

2.2 STEP 2: TASK SELECTION

To ensure that CONTEXTBENCH evaluates challenging and non-trivial context retrieval scenarios, we rank candidate tasks using three complementary difficulty metrics, *i.e.*, agent solvability, the scope and dispersion of edits in the ground-truth patch.

Agent Solvability. We automatically crawl results from the public leaderboards of the source benchmarks and use them as a coarse proxy for task difficulty. For each task, we record the number of agents that successfully resolve it and prioritize tasks that remain unsolved or have been solved by only a few agents.

Edit Scope. For each issue task, we analyze its ground-truth patch and record the number of modified files. We prioritize tasks with larger edit scopes, as they typically require more extensive context retrieval and involve more complex reasoning patterns.

Edit Dispersion. We measure the distribution of code edits across files and directories in the repository. To quantify this, we compute the average structural distance between edited regions in the repository tree parsed by `Tree-Sitter`. We prioritize tasks with highly dispersed edits that span multiple modules or distant files, since these cases involve broader contextual dependencies over scattered code regions.

Based on these difficulty metrics, we select 1,500 candidate tasks. We then manually review them to remove cases that appear challenging by metrics but are semantically trivial, such as large-scale

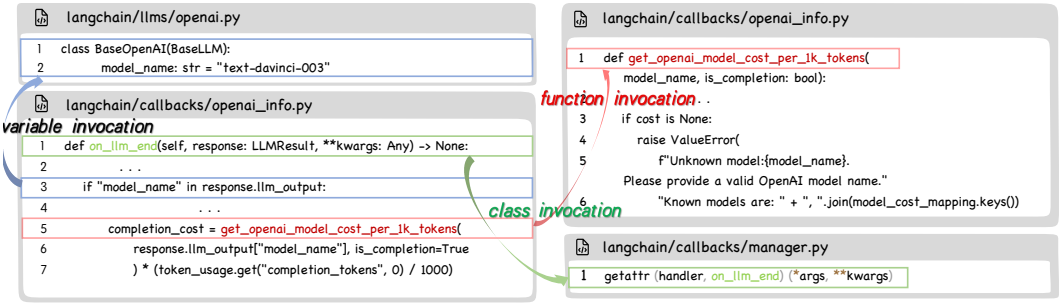


Figure 2: **An illustrative example of the human-verified gold context.** This example is derived from the issue task associated with Pull Request #4009 in the `langchain-ai/langchain` repository. The blue, red, and green arrows represent the manually annotated contextual dependencies associated with variable, function, and class invocations, respectively.

variable renaming across multiple files or bulk formatting changes that do not affect program logic, resulting in a final set of 1,136 tasks. More implementation details are provided in Appendix B.

2.3 Step 3: Expert Annotation

The annotation is conducted collaboratively by six authors of this paper and a group of engaged expert developers over four months. To ensure quality and consistency, we adopt a *human-in-the-loop* procedure where annotators iteratively construct and refine gold contexts for 1,136 selected tasks, validating them through LLM-based patch generation and inter-annotator agreement checks. When executable verification fails, agreement between two independent annotations and expert adjudication guide acceptance. All annotators have over three years of software development experience working with large-scale codebases. To ensure the annotators can proficiently mark the data, we provide detailed guidelines and examples that illustrate how to trace code dependencies, identify relevant artifacts, and record annotations in a standardized format. We also specify clear criteria and task requirements to guide annotators throughout the process. Please refer to the details of our annotation process in Appendix F.

We define gold context as a compact set of repository source-code regions that experts identify as sufficient for the issue, supported by executable verification or inter-annotator review. It is a reference for retrieval evaluation, without a guarantee of global minimality; the formal definition and scope are given in Appendix D.

Context Annotation. The annotation process begins with the ground-truth patch for each issue task. Annotators follow the provided guidelines to trace essential code dependencies and semantically related artifacts originating from the modified regions. For each edited location, they inspect its function and class invocations, inheritance relations, and related control-flow and data-flow paths. Annotators also review the surrounding code within the same file or module to identify components that are semantically relevant to the modification. Through this tracing process, annotators are instructed to ensure that the recorded contexts are sufficient for issue resolution while remaining as compact as possible by excluding redundant or irrelevant code.

Context Verification. To validate whether each annotated context is sufficient to resolve its corresponding issue, we use GPT-5 as a single, fixed verifier. For each task, the model is conditioned solely on the annotated context and prompted to generate five candidate patches. We then consider an annotated context sufficient if at least one generated patch successfully passes the official test suite, including both fail-to-pass and pass-to-pass test cases. One successful attempt establishes operational feasibility for this verifier, which is the aim of the check; failure does not establish that the context is insufficient and instead triggers further annotation and review. Detailed rationale and protocol are provided in Appendix G.

Context Refinement. When an annotated context passes verification, we assign it to a different annotator, distinct from the one who performed the initial annotation, to conduct compactness checking. This secondary annotator reviews the verified context to identify and remove any code

segment that is redundant or irrelevant to issue resolution. The two annotators jointly review the updated context, resolve disagreements, and agree on a consensus version of the context.

If executable verification fails, we obtain two independent context annotations and compute their Jaccard similarity at the line level before joint review. If the similarity is at least 0.8 and the experts reach consensus, we retain the consensus context. When the similarity is below 0.8, a third expert joins the discussion to adjudicate disagreements.

In both branches, after refinement and consensus review, we re-run the verification procedure on the updated context before finalizing the gold context.

Consequently, the annotation process produces a gold context represented by a compact set of code regions that are essential for resolving the issue and generating the corresponding patch, as illustrated in Figure 2. This collaborative refinement aims to preserve sufficient evidence while removing clearly redundant code; it does not establish a globally minimal context.

2.4 CONTEXTBENCH STATISTICS

We develop a systematic data filtering pipeline to collect human-verified gold contexts through deduplication, selection, and expert annotation (detailed in Appendix A). As summarized in Table 1, CONTEXTBENCH comprises 66 repositories and 1,136 issue tasks across eight programming languages, with context annotations provided at the file, block, and line levels, encompassing 4,548 files, 23,116 blocks, and 522,115 lines of code. These gold contexts enable systematic evaluation of intermediate context retrieval beyond final task resolution rates. For efficient evaluation, we construct CONTEXTBENCH Lite by selecting the 500 tasks with the smallest summed ranks across agent solvability, edit scope, and edit dispersion (Appendix B). The context-length distributions are shown in Figure 4 in Appendix D.

Table 1: **Statistics of CONTEXTBENCH.** Task and gold-context counts across eight programming languages.

Language	#Repo	#Task	Context Statistics		
			#File	#Block	#Line
Python	20	512	1,520	6,714	115,122
Java	6	57	262	3,030	49,057
JavaScript	9	153	819	3,949	87,907
TypeScript	8	119	537	1,106	40,621
Go	7	104	679	3,000	71,596
Rust	9	63	272	1,842	50,402
C	3	68	250	1,591	62,300
C++	4	60	209	1,884	45,110
Total	**66**	**1,136**	**4,548**	**23,116**	**522,115**

Scope of Gold Context. Our gold contexts consist solely of source-code regions required for issue resolution. Issue descriptions, execution traces, stack traces, tests, and documentation provide auxiliary diagnostic evidence that helps an agent locate the relevant implementation in the repository. The final patch-context declaration and its precision, recall, and F1 scores concern implementation source code. Additional implementation context outside the annotated reference can lower precision even when it assists the agent; readers should interpret precision relative to this reference scope.

2.5 CONTEXTBENCH EVALUATION PROTOCOL

To evaluate the context retrieval ability of coding agents, we instrument their execution to trace the code regions they inspect during issue resolution and subsequently compare their retrieved contexts with human-verified gold contexts.

Agent Context Tracing. For each issue task, we run the full resolution process and parse tool commands and their returned code content to record inspected source-code regions, including file paths and line ranges. These command-derived records form the intermediate context snapshots along the trajectory, including code lines returned by search commands such as `grep`. Explored context is their union over the complete run, deduplicated across steps (Appendix I). Before final submission, we separately request a `<PATCH_CONTEXT>` declaration containing the source-code regions the agent identifies as required to generate its patch. The declaration provides a common measure of final context selection across pipeline and iterative agents. We score the declared context against the gold reference and use the command-derived observations to measure retrieval dynamics and evidence drop between exploration and final selection. More implementation details are provided in Appendix C.

Agent Context Evaluation. We then parse the entire repository with the `tree-sitter` tool to construct a unified structural coordinate system, onto which both the gold and agent contexts

are mapped for comparison. Each agent context is aligned with the gold reference across three granularities: (i) the file level, by matching file paths; (ii) the block level, by aligning *Abstract Syntax Tree* (AST) nodes at `definition` level, including classes, functions, interfaces, and trait declarations (detailed in Appendix E); and (iii) the line level, by comparing line ranges using their byte offsets within files. This structured alignment enables fine-grained comparison between the gold and agent contexts. Based on this alignment, we evaluate the final declared context using recall, precision, and F1 at the file, block, and line levels, and the cumulative explored context using line-level F1. We report process-level efficiency, redundancy, and evidence drop at the line level, with detailed definitions provided in Appendix H. We separately report Pass@1 for final patch correctness under each task's official test suite; unlike retrieval F1, it measures issue resolution rather than context overlap.

Table 2: **Performance of different coding agents on context retrieval.** Evaluated on CONTEXTBENCH Lite with GPT-5. R/P denote recall/precision. Explored F1 measures line-level overlap between the cumulative observed source-code context and the gold context; the other retrieval metrics evaluate the final declared patch context. Pass@1 measures issue resolution. Full explored-context results appear in Appendix I.

Coding Agent	File: Patch			Block: Patch			Line: Patch			Explored F1↑	Pass@1↑
	R↑	P↑	F1↑	R↑	P↑	F1↑	R↑	P↑	F1↑		
Agentless	0.609	0.352	0.390	0.328	0.344	0.242	0.461	**0.318**	**0.376**	**0.357**	0.452
mini-SWE-Agent	0.682	**0.709**	0.634	0.645	**0.369**	0.375	0.606	0.301	0.312	0.296	0.472
SWE-agent	0.726	0.537	0.544	0.625	0.312	0.285	0.476	0.228	0.208	0.198	0.490
OpenHands	0.733	0.400	0.463	0.505	0.283	0.190	0.472	0.203	0.130	0.124	0.490
Prometheus	0.717	0.336	0.403	0.646	0.258	0.285	0.584	0.195	0.231	0.219	0.512
Codex	0.728	0.612	**0.649**	0.658	0.358	**0.401**	0.617	0.293	0.321	0.305	0.522
Claude Code	**0.741**	0.583	0.642	**0.671**	0.347	0.394	**0.623**	0.281	0.327	0.311	**0.534**

3 CAN CODING AGENTS RETRIEVE EFFECTIVE CONTEXTS TO RESOLVE ISSUES?

3.1 RQ1: BENCHMARKING CODING AGENTS

Table 2 benchmarks seven widely used and actively maintained coding agents: Agentless (Xia et al., 2024), mini-SWE-agent, SWE-agent (Yang et al., 2024), OpenHands (Wang et al., 2025), Prometheus (Pan et al., 2026), Codex (OpenAI, 2026a), and Claude Code (Anthropic, 2026a). All agents share the GPT-5 backbone (Singh et al., 2025), evaluation tasks, and scoring protocol. The unit of comparison is each agent's **complete context retrieval mechanism**: its tools, prompts, search strategy, context management, and stopping decisions jointly determine the context it retrieves. Experimental details and comparison scope are provided in Appendices C.2 and M.

The results reveal distinct context retrieval behaviors across coding agents, shaped by their underlying retrieval mechanisms. Agentless uses embedding-based, staged localization and obtains the highest patch-context line-level precision (0.318) and F1 (0.376), but the lowest recall at every granularity (0.609 at the file level, 0.328 at the block level, and 0.461 at the line level). SWE-agent and OpenHands expose specialized interfaces for iterative exploration, while Prometheus uses graph-based navigation. These agents retrieve broader context than Agentless, but their lower line-level precision shows that the additional coverage also includes code outside the gold reference.

The shell-based agents (mini-SWE-agent, Claude Code, and Codex) achieve the three highest patch-context file-level F1 scores (0.634 to 0.649) and block-level F1 scores (0.375 to 0.401). Claude Code and Codex also obtain the two highest Pass@1 scores (0.534 and 0.522). Their retrieval relies on general-purpose shell operations such as `grep`, `find`, and `sed`, which let the model inspect results and adapt its next query. The lightweight mini-SWE-agent ranks immediately behind them in F1 at the file and block levels, ahead of more elaborate scaffolds using embeddings, specialized interfaces, or graphs.

Explored-context F1 yields the same agent rankings as patch-context F1 at all three granularities (Appendix I). Together, these results suggest that additional retrieval machinery does not consistently improve context quality. Specialized methods can improve individual metrics, as Agentless does at the line level, while flexible shell-based exploration achieves the strongest overall retrieval at the file and block levels and downstream resolution in this comparison. This pattern echoes the

"Bitter Lesson": general model reasoning with simple tools can be more effective than hand-designed retrieval procedures, suggesting potential over-engineering in the evaluated scaffolds.

Answer to RQ1: Direct shell-based retrieval is sufficient and achieves the strongest overall performance. More elaborate retrieval mechanisms using embeddings, specialized interfaces, or graphs do not consistently improve context quality, illustrating the "Bitter Lesson" and suggesting that current retrieval scaffolds are over-engineered.

Table 3: **Performance of various LLMs on code context retrieval.** Evaluated on CONTEXTBENCH Lite with mini-SWE-agent. R/P denote recall/precision. Explored F1 measures line-level overlap between the cumulative observed source-code context and the gold context; the other retrieval metrics evaluate the final declared patch context. Pass@1 measures issue resolution. Full explored-context results appear in Appendix I.

LLM	File: Patch			Block: Patch			Line: Patch			Explored F1↑	Pass@1↑
	R↑	P↑	F1↑	R↑	P↑	F1↑	R↑	P↑	F1↑		
Gemini 2.5 Pro	0.587	0.752	0.600	0.393	**0.632**	0.403	0.313	0.529	0.311	0.295	0.364
Devstral 2	0.660	0.693	0.615	0.478	0.576	0.422	0.404	0.485	0.332	0.315	0.402
Gemini 3.1 Pro	0.612	0.647	0.621	0.443	0.583	0.428	0.358	**0.541**	0.329	0.313	0.422
GPT-5	0.682	0.709	0.634	0.645	0.369	0.375	0.606	0.301	0.312	0.296	0.472
Claude Sonnet 4.5	0.720	0.665	0.624	0.631	0.449	0.420	0.588	0.374	0.344	0.327	0.530
GPT-5.4	0.812	0.764	**0.736**	0.712	0.483	0.461	0.634	0.358	0.347	0.330	0.558
Claude Opus 4.7	**0.847**	**0.781**	0.728	**0.724**	0.527	**0.482**	**0.638**	0.401	**0.358**	**0.340**	**0.592**

3.2 RQ2: BENCHMARKING LARGE LANGUAGE MODELS

Table 3 benchmarks seven state-of-the-art LLMs, including GPT-5 (Singh et al., 2025), Claude Sonnet 4.5 (Anthropic, 2025), Gemini 2.5 Pro (Comanici et al., 2025), Devstral 2 (Mistral AI, 2025), and GPT-5.4 (OpenAI, 2026b), Claude Opus 4.7 (Anthropic, 2026b), and Gemini 3.1 Pro (Google Gemini Team, 2026), in retrieving relevant code contexts from large-scale codebases when resolving issues using the standard coding agent scaffold (*i.e.*, mini-SWE-agent (Yang et al., 2024)). Detailed experiment settings are listed in Section C.1.

The results suggest that state-of-the-art LLMs still face challenges in retrieving effective context during issue resolution. Their patch-context block-level F1 scores range from 0.375 to 0.482, while patch-context line-level F1 scores range from 0.311 to 0.358. Moreover, higher recall does not necessarily indicate better context retrieval performance, as a clear trade-off exists between recall and precision. Some LLMs tend to expand retrieved contexts aggressively to achieve higher recall, while introducing excessive irrelevant content that results in lower precision. For example, GPT 5 achieves higher recall at both the block and line levels than Claude Sonnet 4.5, but lower precision and F1.

We also observe a strong positive correlation between mean patch-context recall and Pass@1 across the seven evaluated models (Pearson $r = 0.921$ at the line level, computed from the reported means in Table 3). LLM agents that cover a larger portion of gold contexts tend to achieve higher Pass@1 scores. This finding reinforces the central premise of CONTEXTBENCH: context retrieval quality is not merely a diagnostic auxiliary metric but a meaningful predictor of end-to-end performance. The tradeoff between recall and precision also highlights the importance of evaluating coverage and evidence selection together.

Answer to RQ2: State-of-the-art LLMs struggle with effective code context retrieval, often covering relevant information at the cost of introducing substantial noise and sacrificing retrieval precision.

3.3 RQ3: DOES BETTER RETRIEVAL LEAD TO BETTER RESOLUTION?

Table 4 reports an oracle study examining how access to gold context affects issue resolution across five coding agents. For each agent, we compare its standard retrieval-based Pass@1 with an oracle setting that supplies gold context at the start and asks the agent to generate a patch directly from that context.

Table 4: **Oracle study: Pass@1 gains.**

Agent	w/o Gold	w/ Gold	Δ
mini-SWE-agent	0.472	**0.702**	+0.230
SWE-agent	0.490	**0.698**	+0.208
Agentless	0.452	**0.674**	+0.222
Claude Code	0.534	**0.728**	+0.194
Codex	0.522	**0.716**	+0.194

Providing gold context improves Pass@1 by *19.4–23.0 percentage points* across all five agents. These gains across open-source and proprietary agents indicate that context retrieval remains a major bottleneck in issue resolution.

In a complementary setting, we prompt GPT 5 directly, without an agent, to generate a patch using only the issue description and the bug snippet, with no additional repository context. The resolution rate is only 1.2%, supporting the need to retrieve additional repository context for issue resolution.

Answer to RQ3: Context retrieval remains a major bottleneck in issue resolution. Access to the relevant repository context substantially improves agents' ability to resolve issues.

3.4 RQ4: Analysis of Context Retrieval Strategies and Dynamics

Table 5 compares retrieval strategies across seven LLMs using mini-SWE-agent. We measure steps per instance, lines per step, cost, efficiency, redundancy, and evidence drop to characterize context acquisition and final selection. Metric definitions are provided in Appendix H.

Table 5: **Context retrieval strategies and dynamics across LLMs.** Efficiency, redundancy, and drop use source-code lines; arrows indicate higher/lower is better.

LLM	Steps /Inst.	Lines /Step	Cost ($)	Eff.↑	Red.↓	Drop↓
GPT-5	**5.87**	119.29	0.45	0.591	0.487	0.179
GPT-5.4	6.21	**128.45**	0.58	0.647	**0.462**	**0.158**
Claude Sonnet 4.5	14.38	29.74	0.76	0.658	0.708	0.196
Claude Opus 4.7	15.62	32.18	**1.24**	**0.712**	0.685	0.171
Gemini 2.5 Pro	7.57	26.29	**0.38**	0.529	0.558	0.431
Gemini 3.1 Pro	8.13	31.45	0.52	0.572	0.541	0.378
Devstral 2	**22.16**	**11.98**	0.91	0.616	0.672	0.435

GPT-5 uses the fewest steps with relatively large contexts, whereas Devstral 2 uses the most steps with the smallest contexts. The two Claude models fall between these extremes, with Opus achieving the highest patch-context line-level F1 and Pass@1 in Table 3. Cost does not follow step count alone: Devstral 2 costs more than GPT-5, while Opus costs the most despite using fewer steps than Devstral 2.

Efficiency and redundancy reveal distinct aspects of retrieval. Opus achieves the highest efficiency, while Sonnet combines high efficiency with the highest redundancy. Efficient acquisition of gold context can therefore coexist with substantial repeated retrieval.

All models exhibit positive evidence drop: some observed gold-relevant context is absent from the final patch-context declaration. Devstral 2 and Gemini 2.5 Pro have the largest drops and the two lowest Pass@1 scores. This gap identifies final context selection as a further dimension of retrieval behavior beyond acquisition efficiency and redundancy.

Answer to RQ4: LLMs can acquire relevant code efficiently yet still retrieve substantial duplicate content and leave relevant evidence out of their final context selection.

3.5 RQ5: Analysis of Gold Context Robustness

Our gold contexts are human-annotated based on a gold patch. However, in practice, a single issue may admit multiple semantically equivalent but syntactically different patches, raising the question of whether the collected gold context is reliable under solution multiplicity.

We analyze 331 tasks from ContextBench with at least two distinct test-passing patches available per task (884 patches in total). Alternative patches are collected from our own agent runs and public experimental releases associated with benchmark leaderboards. For each patch, we derive a patch-conditioned gold context using the same context tracing procedure. We quantify consistency using the Jaccard similarity between each pair of contexts, detailed in Appendix H. For each task, we report the average similarity across all context pairs. At the block level (e.g., function/class), the Jaccard similarity is exactly 1.0, indicating identical block sets across the patch-conditioned contexts for each task. At the line level, the average Jaccard similarity is 0.812, indicating that gold contexts remain highly consistent across alternative valid patches. The line-level variation mostly reflects differences in how much of the same code blocks is needed, rather than disagreement on which blocks are relevant. These results confirm that the gold context is robust under solution multiplicity.

Answer to RQ5: Gold contexts remain highly consistent across alternative valid patches, supporting their reliability as references for context retrieval evaluation.

4 Related Work

Repository-Level Code Evaluation Benchmarks. Repository-level benchmarks ground evaluation in real-world codebases and executable test suites, making them the de facto standard for assessing end-to-end issue resolution. SWE-bench and its variants (Lite, Verified, multilingual extensions (Zan et al., 2025), long-horizon tasks (Deng et al., 2025b), live environments (Zhang et al., 2025), goal-oriented settings (Yang et al., 2025a), and freelance scenarios (Miserendino et al., 2025)) evaluate whether a system can produce a test-passing patch under realistic repository settings (Yang et al., 2024; Jimenez et al., 2024), while complementary benchmarks such as RepoBench (Liu et al., 2023) and SWE-PolyBench (Rashid et al., 2025) emphasize cross-file completion or structure-aware analysis. Despite their realism, these benchmarks are predominantly *outcome-driven*: they largely measure final task success, offering limited visibility into whether agents succeed by identifying the right information or by compensating through extensive exploration (Pan et al., 2026). Our work complements outcome-only evaluation by (i) introducing *verified gold contexts* and (ii) enabling dynamic, process-level evaluation of context retrieval beyond final outcomes.

Coding Agents. LLM-based coding agents address repository-level issue resolution through either pipeline-based or agentic designs. Pipeline-based systems decompose tasks into fixed stages with controlled context selection (Xia et al., 2024; Zhang et al., 2024; Ruan et al., 2024; Gao et al., 2025; Yang et al., 2025b), agentic systems adopt open-ended interaction loops in which agents dynamically search, inspect, and modify codebases via tool calls (Yang et al., 2024; Wang et al., 2025; Xia et al., 2025; Chen et al., 2025a; Pan et al., 2026). Despite architectural diversity, context retrieval remains a shared bottleneck. Existing benchmarks primarily evaluate end-to-end success, making it difficult to compare how agents retrieve and manage code context. ContextBench fills this gap by providing a retrieval-centric benchmark that directly evaluates context acquisition behavior, independent of final patch correctness.

Trajectory Analysis and Failure Diagnosis. Recent work analyzes agent trajectories to diagnose failure modes via manual inspection, error taxonomies, or large-scale statistics over execution logs (Zhu et al., 2025; Liu et al., 2025; Chen et al., 2025b; Bouzenia & Pradel, 2025; Cemri et al., 2025; Pathak et al., 2025). Such analyses have improved understanding of where agents fail (e.g., reasoning deadlocks, tool misuse, runtime exceptions), and some studies report that context-related errors are frequent in repository-level settings. However, trajectory analysis is often labor-intensive, difficult to standardize across frameworks, and typically provides *post hoc* explanations rather than instance-level supervision. ContextBench addresses this gap by offering a scalable, standardized evaluation layer: verified gold contexts and automatic metrics that quantify retrieval effectiveness without requiring manual reading of long trajectories, while still enabling connections between intermediate retrieval behavior and end-to-end outcomes.

5 Conclusion

We introduced ContextBench, a benchmark for process-level evaluation of context retrieval in LLM-based coding agents. By augmenting existing issue-resolution benchmarks with human-annotated gold contexts and retrieval-centric metrics, ContextBench enables fine-grained analysis beyond end-to-end success rates. Our evaluation reveals limited gains from complex agent scaffolding, higher recall than precision for most LLMs at the block and line levels, and substantial gaps between retrieved and utilized context. These findings highlight the importance of process-oriented evaluation and intermediate retrieval signals for improving LLM-based software engineering systems. ContextBench provides a new evaluation paradigm for studying agent reasoning and context utilization in large-scale codebases. We hope it supports the development of more reliable, interpretable, and trustworthy coding agents for real-world software engineering.

AI USE STATEMENT

Generative AI was used to provide feedback on the manuscript's methodology and interpretation of results, check consistency, assist with wording and LaTeX editing, and repair figure exports. The LLM-based context verification and agent evaluations are described in Section 2.3 and the experimental sections. The authors are responsible for the final text, claims, and artifacts.

REFERENCES

Anthropic. Introducing claude sonnet 4.5. https://www.anthropic.com/news/claude-sonnet-4-5, 2025. Accessed: 2026-01-31.

Anthropic. Claude Code: Overview. https://code.claude.com/docs/en/overview, 2026a. Documentation. Accessed: 2026-09-20.

Anthropic. Introducing Claude Opus 4.7. https://www.anthropic.com/news/claude-opus-4-7, April 2026b. Accessed: 2026-09-20.

Islem Bouzenia and Michael Pradel. Understanding software engineering agents: A study of thought-action-result trajectories. *arXiv preprint arXiv:2506.18824*, 2025.

Mert Cemri, Melissa Z Pan, Shuyi Yang, Lakshya A Agrawal, Bhavya Chopra, Rishabh Tiwari, Kurt Keutzer, Aditya Parameswaran, Dan Klein, Kannan Ramchandran, et al. Why do multi-agent llm systems fail? *arXiv preprint arXiv:2503.13657*, 2025.

Silin Chen, Shaoxin Lin, Xiaodong Gu, Yuling Shi, Heng Lian, Longfei Yun, Dong Chen, Weiguo Sun, Lin Cao, and Qianxiang Wang. Swe-exp: Experience-driven software issue resolution. *arXiv preprint arXiv:2507.23361*, 2025a.

Zhi Chen, Wei Ma, and Lingxiao Jiang. Beyond final code: A process-oriented error analysis of software development agents in real-world github scenarios. *arXiv preprint arXiv:2503.12374*, 2025b.

Zhaoyang Chu, Jiarui Hu, Xingyu Jiang, Pengyu Zou, Han Li, Chao Peng, Peter O'Hearn, Earl T Barr, Mark Harman, Federica Sarro, et al. Terminalworld: Benchmarking agents on real-world terminal tasks. *arXiv preprint arXiv:2605.22535*, 2026.

Gheorghe Comanici, Eric Bieber, Mike Schaekermann, Ice Pasupat, Noveen Sachdeva, Inderjit Dhillon, Marcel Blistein, Ori Ram, Dan Zhang, Evan Rosen, et al. Gemini 2.5: Pushing the frontier with advanced reasoning, multimodality, long context, and next generation agentic capabilities. *arXiv preprint arXiv:2507.06261*, 2025.

Le Deng, Zhonghao Jiang, Jialun Cao, Michael Pradel, and Zhongxin Liu. Nocode-bench: A benchmark for evaluating natural language-driven feature addition. *arXiv preprint arXiv:2507.18130*, 2025a.

Xiang Deng, Jeff Da, Edwin Pan, Yannis Yiming He, Charles Ide, Kanak Garg, Niklas Lauffer, Andrew Park, Nitin Pasari, Chetan Rane, et al. Swe-bench pro: Can ai agents solve long-horizon software engineering tasks? *arXiv preprint arXiv:2509.16941*, 2025b.

Pengfei Gao, Zhao Tian, Xiangxin Meng, Xinchen Wang, Ruida Hu, Yuanan Xiao, Yizhou Liu, Zhao Zhang, Junjie Chen, Cuiyun Gao, et al. Trae agent: An llm-based agent for software engineering with test-time scaling. *arXiv preprint arXiv:2507.23370*, 2025.

Google Gemini Team. Gemini 3.1 Pro: A smarter model for your most complex tasks. https://blog.google/innovation-and-ai/models-and-research/gemini-models/gemini-3-1-pro/, February 2026. Accessed: 2026-09-20.

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? In *International Conference on Learning Representations*, 2024. URL https://arxiv.org/abs/2310.06770.

Simiao Liu, Fang Liu, Liehao Li, Xin Tan, Yinghao Zhu, Xiaoli Lian, and Li Zhang. An empirical study on failures in automated issue solving. *arXiv preprint arXiv:2509.13941*, 2025.

Tianyang Liu, Canwen Xu, and Julian McAuley. Repobench: Benchmarking repository-level code auto-completion systems. *arXiv preprint arXiv:2306.03091*, 2023. URL https://arxiv.org/abs/2306.03091.

Mike A. Merrill, Alexander G. Shaw, Nicholas Carlini, Boxuan Li, Harsh Raj, Ivan Bercovich, Lin Shi, Jeong Yeon Shin, Thomas Walshe, E. Kelly Buchanan, Junhong Shen, Guanghao Ye, Haowei Lin, Jason Poulos, Maoyu Wang, Marianna Nezhurina, Jenia Jitsev, Di Lu, Orfeas Menis Mastromichalakis, Zhiwei Xu, Zizhao Chen, Yue Liu, Robert Zhang, Leon Liangyu Chen, Anurag Kashyap, Jan-Lucas Uslu, Jeffrey Li, Jianbo Wu, Minghao Yan, Song Bian, Vedang Sharma, Ke Sun, Steven Dillmann, Akshay Anand, Andrew Lanpouthakoun, Bardia Koopah, Changran Hu, Etash Guha, Gabriel H. S. Dreiman, Jiacheng Zhu, Karl Krauth, Li Zhong, Niklas Muennighoff, Robert Amanfu, Shangyin Tan, Shreyas Pimpalgaonkar, Tushar Aggarwal, Xiangning Lin, Xin Lan, Xuandong Zhao, Yiqing Liang, Yuanli Wang, Zilong Wang, Changzhi Zhou, David Heineman, Hange Liu, Harsh Trivedi, John Yang, Junhong Lin, Manish Shetty, Michael Yang, Nabil Omi, Negin Raoof, Shanda Li, Terry Yue Zhuo, Wuwei Lin, Yiwei Dai, Yuxin Wang, Wenhao Chai, Shang Zhou, Dariush Wahdany, Ziyu She, Jiaming Hu, Zhikang Dong, Yuxuan Zhu, Sasha Cui, Ahson Saiyed, Arinbjörn Kolbeinsson, Jesse Hu, Christopher Michael Rytting, Ryan Marten, Yixin Wang, Alex Dimakis, Andy Konwinski, and Ludwig Schmidt. Terminal-bench: Benchmarking agents on hard, realistic tasks in command line interfaces. *arXiv preprint arXiv:2601.11868*, 2026.

Samuel Miserendino, Michele Wang, Tejal Patwardhan, and Johannes Heidecke. Swe-lancer: Can frontier llms earn $1 million from real-world freelance software engineering?, 2025. URL https://arxiv.org/abs/2502.12115.

Mistral AI. Introducing: Devstral 2 and mistral vibe cli. https://mistral.ai/news/devstral-2-vibe-cli, 2025. Accessed: 2026-01-31.

OpenAI. Codex CLI. https://developers.openai.com/codex/cli/, 2026a. Documentation. Accessed: 2026-09-20.

OpenAI. GPT-5.4. https://developers.openai.com/api/docs/models/gpt-5.4, 2026b. Model documentation. Accessed: 2026-09-20.

Jiayi Pan, Xingyao Wang, Graham Neubig, Navdeep Jaitly, Heng Ji, Alane Suhr, and Yizhe Zhang. Training software engineering agents and verifiers with SWE-gym. In *Forty-second International Conference on Machine Learning*, 2025.

Yue Pan, Zimin Chen, Siyu Lu, Zhaoyang Chu, Xiang Li, Han Li, Yang Feng, Claire Le Goues, Federica Sarro, Martin Monperrus, and He Ye. Prometheus: Towards long-horizon codebase navigation for repository-level problem solving, 2026. URL https://arxiv.org/abs/2507.19942v2.

Divya Pathak, Harshit Kumar, Anuska Roy, Felix George, Mudit Verma, and Pratibha Moogi. Detecting silent failures in multi-agentic ai trajectories. *arXiv preprint arXiv:2511.04032*, 2025.

Muhammad Shihab Rashid, Christian Bock, Yuan Zhuang, Alexander Buchholz, Tim Esler, Simon Valentin, Luca Franceschi, Martin Wistuba, Prabhu Teja Sivaprasad, Woo Jung Kim, Anoop Deoras, Giovanni Zappella, and Laurent Callot. Swe-polybench: A multi-language benchmark for repository level evaluation of coding agents. *arXiv preprint arXiv:2504.08703*, 2025. URL https://arxiv.org/abs/2504.08703.

Haifeng Ruan, Yuntong Zhang, and Abhik Roychoudhury. Specrover: Code intent extraction via llms. *arXiv preprint arXiv:2408.02232*, 2024.

Aaditya Singh, Adam Fry, Adam Perelman, Adam Tart, Adi Ganesh, Ahmed El-Kishky, Aidan McLaughlin, Aiden Low, AJ Ostrow, Akhila Ananthram, et al. Openai gpt-5 system card. *arXiv preprint arXiv:2601.03267*, 2025.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, Hoang H. Tran, Fuqiang Li, Ren Ma, Mingzhang Zheng, Bill Qian, Yanjun Shao, Niklas Muennighoff, Yizhe Zhang, Binyuan Hui, Junyang Lin, and et al. Openhands: An open platform for AI software developers as generalist agents. In *The Thirteenth International Conference on Learning Representations, ICLR 2025, Singapore, April 24-28, 2025*. OpenReview.net, 2025.

Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. Agentless: Demystifying llm-based software engineering agents. *arXiv preprint arXiv:2407.01489*, 2024. URL https://arxiv.org/abs/2407.01489.

Chunqiu Steven Xia, Zhe Wang, Yan Yang, Yuxiang Wei, and Lingming Zhang. Live-swe-agent: Can software engineering agents self-evolve on the fly? *arXiv preprint arXiv:2511.13646*, 2025.

John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik R Narasimhan, and Ofir Press. SWE-agent: Agent-computer interfaces enable automated software engineering. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024.

John Yang, Kilian Lieret, Joyce Yang, Carlos E. Jimenez, Ofir Press, Ludwig Schmidt, and Diyi Yang. Codeclash: Benchmarking goal-oriented software engineering, 2025a. URL https://arxiv.org/abs/2511.00839.

Xu Yang, Jiayuan Zhou, Michael Pacheco, Wenhan Zhu, Pengfei He, Shaowei Wang, Kui Liu, and Ruiqi Pan. Lingxi: Repository-level issue resolution framework enhanced by procedural knowledge guided scaling. *arXiv preprint arXiv:2510.11838*, 2025b.

Daoguang Zan, Zhirong Huang, Wei Liu, Hanwu Chen, Linhao Zhang, Shulin Xin, Lu Chen, Qi Liu, Xiaojian Zhong, Aoyan Li, Siyao Liu, Yongsheng Xiao, Liangqiang Chen, Yuyu Zhang, Jing Su, Tianyu Liu, Rui Long, Kai Shen, and Liang Xiang. Multi-SWE-bench: A multilingual benchmark for issue resolving. *arXiv preprint arXiv:2504.02605*, 2025. URL https://arxiv.org/abs/2504.02605.

Linghao Zhang, Shilin He, Chaoyun Zhang, Yu Kang, Bowen Li, Chengxing Xie, Junhao Wang, Maoquan Wang, Yufan Huang, Shengyu Fu, Elsie Nallipogu, Qingwei Lin, Yingnong Dang, Saravan Rajmohan, and Dongmei Zhang. Swe-bench goes live!, 2025. URL https://arxiv.org/abs/2505.23419.

Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. Autocoderover: Autonomous program improvement. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2024, pp. 1592–1604, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400706127.

Kunlun Zhu, Zijia Liu, Bingxuan Li, Muxin Tian, Yingxuan Yang, Jiaxun Zhang, Pengrui Han, Qipeng Xie, Fuyang Cui, Weijia Zhang, et al. Where llm agents fail and how they can learn from failures. *arXiv preprint arXiv:2509.25370*, 2025.

A Details on Data Filtering

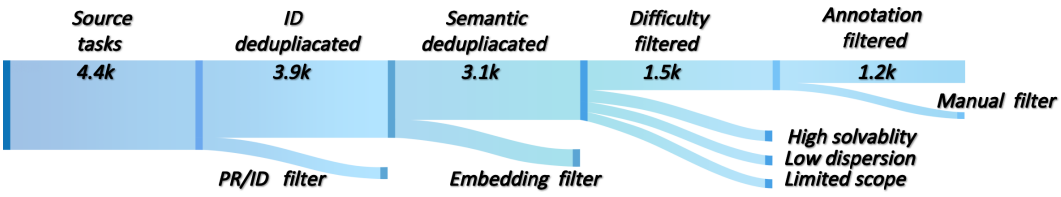


Figure 3: **The data flow during the ContextBench construction pipeline.** We construct ContextBench step by step, starting from task deduplication and selection to gold context annotation.

This subsection provides a detailed description of the data filtering pipeline used to construct the final evaluation task set. As shown in Figure 3, the filtering process removes duplicate and semantically trivial tasks while prioritizing challenging context retrieval scenarios before expert annotation.

Initial task pool. We begin with 4,497 source tasks from SWE-bench Verified, Multi-SWE-bench, SWE-PolyBench PB500, and SWE-Bench Pro. Although these tasks originate from real-world issue and pull request scenarios, our work does not directly crawl raw repositories; instead, it consolidates and refines tasks from established benchmarks. At this stage, the task set intentionally serves as a broad superset and contains substantial redundancy and variability in task characteristics.

ID-based deduplication. As a first step, we apply an ID-level deduplication procedure that removes tasks with identical or invalid issue/PR identifiers. This step eliminates exact duplicates introduced during dataset aggregation and ensures that each remaining task corresponds to a unique identifier. After this filtering, 3.9k tasks remain.

Semantic deduplication. Next, we perform semantic-level deduplication using an embedding-based filtering strategy. Each task is encoded into a semantic vector representation using a pretrained embedding model, and pairwise cosine similarity is computed between task embeddings. Tasks whose cosine similarity exceeds a predefined threshold of 0.90 are considered semantically redundant and are filtered out, even if their identifiers differ. This step removes near-duplicate tasks that differ only in superficial wording while preserving semantically distinct problem instances. After semantic deduplication, the task set is reduced to 3.1k tasks.

B Details on Task Selection

Difficulty-based selection. After deduplication, we rank candidate tasks using the three signals defined in the main paper. *Agent solvability* is the number of agents that resolve a task in public source-benchmark leaderboard results; we prioritize tasks solved by few or no agents. *Edit scope* is the number of files modified by the ground-truth patch; we prioritize larger edit scopes. *Edit dispersion* is the average structural distance between edited regions in the repository tree parsed with `tree-sitter`; we prioritize edits spread across modules or distant files. These signals select 1,500 candidate tasks before gold-context annotation.

Manual semantic review. We then review the 1,500 candidates and remove tasks that appear difficult under these signals but are semantically trivial, including large-scale variable renaming and bulk formatting changes that do not affect program logic. This review yields 1,136 tasks, which enter the expert annotation, verification, and refinement procedure described in Section 2.3.

Lite subset selection. We rank the 1,136 tasks separately on each of the three difficulty dimensions: the number of successful agents in ascending order, the number of modified files in descending order, and edit dispersion in descending order. Lower ranks indicate greater difficulty under each criterion. We sum the three ranks for each task and select the 500 tasks with the smallest rank sums to form ContextBench Lite. Ties are resolved by giving priority to tasks with a larger edit scope.

C Implementation Details on Agent Context Tracing

We describe our context extraction mechanism and the adaptations applied across multiple agents. Section C.1 covers model variations within mini-SWE-agent; Section C.2 covers cross-agent adaptations for the same model.

C.1 Model Variations within mini-SWE-Agent

C.1.1 Context Extraction Architecture

In this subsection, we analyze how different models behave under the mini-SWE-agent context extraction protocol. We extend the default agent architecture with a `ContextAwareAgent` class to enable structured and standardized context collection.

To enforce strict structure and machine-readability, prompts are augmented with predefined templates, and the `<PATCH_CONTEXT>` block is extracted using a verification function with regular expressions. This setup ensures consistency, reproducibility, and automated verification within mini-SWE-agent workflows.

C.1.2 Two-Stage Submission Protocol

The submission workflow follows a two-stage protocol. In the first stage, when the agent detects either `MINI_SWE_AGENT_FINAL_OUTPUT` or `COMPLETE_TASK_AND_SUBMIT_FINAL_OUTPUT`, it does not finalize the task immediately. Instead, it raises a `ContextRequested` exception, which triggers a structured context request. In the second stage, the agent validates the returned context format and proceeds with final submission only after successful verification.

C.1.3 Format Validation Mechanism

To enforce strict structure and machine-readability, the prompt is augmented with predefined templates that explicitly constrain the output format. Format validation is performed by a dedicated verification function, and a regular expression is used to extract the standardized `<PATCH_CONTEXT>` block for downstream processing.

This design enforces strict structural constraints on contextual information, ensuring consistency, reproducibility, and compatibility with automated verification and downstream processing pipelines in agent-based software engineering workflows.

C.1.4 Cross-Model Evaluation Results

After establishing the context extraction framework within mini-SWE-agent, we evaluated its effectiveness across seven models: GPT-5, Claude Sonnet 4.5, Gemini 2.5 Pro, Devstral 2, GPT-5.4, Claude Opus 4.7, and Gemini 3.1 Pro. Under this standardized extraction protocol, all seven models demonstrated robust adherence to the framework, consistently producing accurate and machine-verifiable `<PATCH_CONTEXT>` information. These results indicate that the context extraction mechanism effectively constrains model outputs, ensuring that the returned context is both structured and semantically relevant for downstream task submission.

During the evaluation, we observed that Devstral 2 occasionally exhibited unexpected behaviors that deviated from the standardized output protocol. These anomalies are discussed in detail in Appendix N, where we provide the corresponding raw interaction traces and highlight the specific issues encountered.

Context Control Templates in mini-SWE-Agent (context_aware.yaml)

```
 context_request_template:  |

<context_request>

Before finalizing your submission, please provide the CODE
CONTEXT you used to generate the patch.
```

Requirements

```
    List ONLY the specific code sections from source files
that you examined to create your patch.
```

Format (strictly follow)

```
    Each entry must include:
- File:  <absolute_file_path>
- Lines:  <start_line>-<end_line>
```

Example

```
    File:  /testbed/src/core/handler.ext
Lines:  34-56

File:  /testbed/lib/utils.ext
Lines:  128-145

File:  /testbed/src/parser.ext
Lines:  67-89
```

DO NOT Include

```
    - Explanations, reasoning, or commentary
- Error messages or debugging output
- Test code or test results
- Documentation or comments
- Any non-code content
```

```
    Please format your response as:

THOUGHT: Providing code context used for patch generation.

<PATCH_CONTEXT>
File:  <absolute_file_path>
Lines:  <start_line>-<end_line>

File:  <absolute_file_path>
Lines:  <start_line>-<end_line>
</PATCH_CONTEXT>
```bash
echo COMPLETE_TASK_AND_SUBMIT_FINAL_OUTPUT && (git status
>/dev/null 2>&1 && git add -A && git diff --cached) || (cd
*/.  2>/dev/null && git add -A && git diff --cached) || (echo
"Error:  No git repository found")
```

CRITICAL: Provide ONLY file paths and line ranges (no
code snippets, no explanations) within <PATCH_CONTEXT> tags.
</context_request>

context_confirmation_template: |
<context_received>
Context received and recorded ({{context_length}} characters).
Now proceeding with final submission.
</context_received>
```

Experiment Setting

```
 environment:
timeout:  180
pull_timeout:  3600
env:
PAGER: cat
MANPAGER: cat
LESS: -R
PIP_PROGRESS_BAR: 'off'
```

```
TQDM_DISABLE: '1'
environment_class:  docker
model:
model_name:  "anthropic/claude-sonnet-4-5-20250929"
model_kwargs:
drop_params:  true
temperature:  0.0
```

C.2 Cross-Agent Comparison for the Same Model

C.2.1 Context Extraction Protocol Extension

To evaluate the impact of agent architecture and prompt design on context extraction, we extended the standardized context enforcement protocol, originally developed for mini-SWE-agent, to six other agents: Agentless, SWE-agent, OpenHands, Prometheus, Codex, and Claude Code. Each of these agents adopts a context extraction framework built upon the mini-SWE-agent as the foundational design, with agent-specific adaptations to accommodate differences in workflow and output formatting.

In all systems, a pre-submission request asks the agent to declare the source-code regions required to generate its patch. The evaluation layer separately parses tool commands and their returned code content to record inspected file paths and line ranges throughout execution. These observations provide the exploration history against which the final patch-context selection is compared when computing evidence drop.

Moreover, prompt templates and verification routines were adjusted for each agent to maintain a uniform, machine-readable `<PATCH_CONTEXT>` structure. These adaptations facilitate a controlled comparison of context extraction effectiveness across heterogeneous agent frameworks, and provide a foundation for analyzing the influence of agent-specific design choices on model performance.

C.2.2 SWE-Agent Implementation

Pre-Submission Context Enforcement Specification in SWE-Agent

Required Format: In the next response, the agent must:

1. Output a `<PATCH_CONTEXT>` block listing the source files and line ranges required to generate the patch.
2. Invoke the `submit` function.

Format for `<PATCH_CONTEXT>`

```
<PATCH_CONTEXT>
File: /testbed/path/to/file1.ext
Lines: 10-50

File: /testbed/path/to/file2.ext
Lines: 100-150
</PATCH_CONTEXT>
```

Rules

- Select the patch-required context from inspected implementation source files (exclude test files).
- Include the exact line ranges required for the patch.
- Do not include code snippets, explanations, or commentary.
- This step is mandatory and cannot be skipped.

config.yaml

```
 model:
name:  gpt-5
api_base:
api_key:
per_instance_cost_limit:  0.0
total_cost_limit:  0.0
temperature:  1.0
top_p:  null
retry:
retries:  10
min_wait:  5
max_wait:  60
completion_kwargs:
timeout:  120
tools:
execution_timeout:  600
env_variables:
PAGER: cat
MANPAGER: cat
LESS: -R
GIT_PAGER: cat
```

C.2.3 OPENHANDS IMPLEMENTATION

Context Extraction Protocol in OpenHands

Command-based exploration tracing. The evaluation layer parses tool commands and their returned source-code content to recover the file paths and line ranges inspected during exploration. This includes code returned by file-reading commands and search commands such as `grep`. The recovered regions are stored as structured exploration records.

Explore Context Format

```
<EXPLORE_CONTEXT>
File: /absolute/path/to/file.ext
Lines: <start>-<end>
</EXPLORE_CONTEXT>
```

Each exploration record contains file-range entries derived from the command and its returned code. Commands returning only paths or status information, such as `ls`, `grep -l`, or `git status`, do not contribute source-code lines. A `grep` command that prints matching code lines contributes those observed lines.

Patch Context Submission Requirement

Before finalizing the submission, the agent must declare the inspected implementation code it identifies as required for patch generation.
Format (strictly enforced):

```
THOUGHT: Providing code context used for patch generation.

<PATCH_CONTEXT>
File: <absolute_file_path>
Lines: <start_line>-<end_line>

File: <absolute_file_path>
Lines: <start_line>-<end_line>
</PATCH_CONTEXT>
```

Only absolute file paths and line ranges are permitted inside the <PATCH_CONTEXT> block. Explanations, comments, debugging output, test code, or any non-code content are strictly disallowed.

Unlike mini-SWE-agent and SWE-Agent, Openhands employs a different prompt format. Nevertheless, this variation in prompt design does not preclude the implementation of an equivalent context extraction mechanism. By adapting the extraction protocol to accommodate the agent-specific prompt structure, we ensure that the resulting <PATCH_CONTEXT> outputs remain consistent, structured, and machine-verifiable, thereby preserving the integrity and comparability of the context extraction process across heterogeneous agents.

C.2.4 AGENTLESS ENHANCEMENTS

Bug Fixes and Compatibility. Additionally, we implement several extensions and bug fixes for Agentless to ensure compatibility with the SWE-bench evaluation harness. We adapt to SWE-bench harness API changes by fixing import paths due to the test_spec module restructuring. We also unify Docker environment configurations by removing hardcoded docker_specs and aligning with build_env_images() to ensure consistent env_image_key generation. Additionally, we remove the deprecated pip option –no-use-pep517 which has been removed in pip $\geq$ 23.0.

Multi-Language Support. To support cross-language evaluation, we optimize the multi-language adaptation capability of Agentless, enabling better support for multi-language benchmarks.

Improvement 1: Dynamic File Extension Adaptation

Strategy: Automatically detect the repository's primary programming language by analyzing file extension frequency, then dynamically inject the most common extension into prompt examples.
Detection Method:

- Traverse repository structure to count file extensions
- Select extension with highest occurrence (e.g., .py, .java, .js, .go)
- Replace hard-coded examples with detected extension

Before: Hard-coded Python Extensions

````
For example:
```
file1.py
file2.py
```
````

````
### Examples:
```
full_path1/file1.py
line: 10
class: MyClass1

full_path2/file2.py
function: MyClass2.my_method
```
````

After: Repository-Adaptive Extensions

````
For example:
```
file1.<detected_extension>
file2.<detected_extension>
```

### Examples:
```
full_path1/file1.<detected_extension>
line: 10
class: MyClass1

full_path2/file2.<detected_extension>
function: MyClass2.my_method
```
````

Impact: Eliminates language-specific bias in fault localization prompts, enabling cross-language applicability without manual configuration.

Improvement 2: Language-Agnostic DIFF Examples

Strategy: Replace language-specific DIFF examples with universal "Hello World" modification patterns that adapt to the detected programming language.

Before: Python-Specific DIFF Example

````
Here is an example:

```python
### mathweb/flask/app.py
<<<<<<< SEARCH
from flask import Flask
=======
import math
from flask import Flask
>>>>>>> REPLACE
```
````

After: Python Hello World Example

````
Here is an example:

```python
### example/hello.py
<<<<<<< SEARCH
print("Hello World")
=======
print("Hello")
print("World")
````
````

````
>>>>>>> REPLACE
```
````

After: Java Hello World Example

````
Here is an example:

```java
### example/Hello.java
<<<<<<< SEARCH
System.out.println("Hello World");
=======
System.out.println("Hello");
System.out.println("World");
>>>>>>> REPLACE
```
````

After: JavaScript Hello World Example

````
Here is an example:

```javascript
### example/hello.js
<<<<<<< SEARCH
console.log("Hello World");
=======
console.log("Hello");
console.log("World");
>>>>>>> REPLACE
```
````

After: Go Hello World Example

````
Here is an example:

```go
### example/hello.go
<<<<<<< SEARCH
fmt.Println("Hello World")
=======
fmt.Println("Hello")
fmt.Println("World")
>>>>>>> REPLACE
```
````

Adaptation Logic:

- Detect repository language from file extensions
- Select corresponding Hello World DIFF template
- Inject into repair prompts at runtime

Impact: Provides language-appropriate syntax examples without domain-specific dependencies (e.g., Flask), improving LLM understanding across diverse codebases.

C.2.5 Framework Generalization Results

After applying the standardized context extraction framework to all seven agents (Agentless, mini-SWE-agent, SWE-agent, OpenHands, Prometheus, Codex, and Claude Code), we observed that each system was able to accurately produce the corresponding <PATCH_CONTEXT> outputs. This demonstrates that the framework is robust and generalizable across heterogeneous agent architectures, effectively constraining model outputs to a structured, machine-verifiable format.
````

The consistent extraction of patch context across agents facilitates reproducibility and auditing, and provides a reliable basis for downstream evaluation of model performance in automated software engineering tasks. Moreover, these results indicate that the underlying principles of context enforcement can be adapted to accommodate differences in prompt design, agent workflow, and system architecture without compromising the fidelity or completeness of the extracted context. Overall, the successful deployment of the framework across multiple agents underscores its utility as a foundational tool for systematic evaluation and comparison of agent-based patch generation systems.

D Gold Context Definition and Scope

Gold Context as a Compact Sufficient Reference. For an issue task T in a fixed repository snapshot, its *gold context* C^G is a finite, human-annotated set of source-code regions, each identified by a repository-relative file path and a code span. Annotators select regions they judge sufficient to understand and construct a correct patch, seeking a compact reference under the annotation guideline. Sufficiency is supported by a test-passing verification attempt when available, or by the multi-annotator review described in Section 2.3; these provide distinct forms of evidence. The reference excludes tests, documentation, issue descriptions, and execution or stack traces, which can assist diagnosis but are outside the implementation context evaluated here. We do not claim global minimality: enforcing a globally minimal sufficient set is generally intractable in repository-scale codebases due to the combinatorial space of dependency chains and code spans. Instead, our refinement stage removes clearly redundant regions while preserving the evidence judged necessary during verification and review (Appendix G). Accordingly, precision/F1 should be interpreted with respect to this compact reference; additional retrieved context can be helpful even if it is absent from C^G.

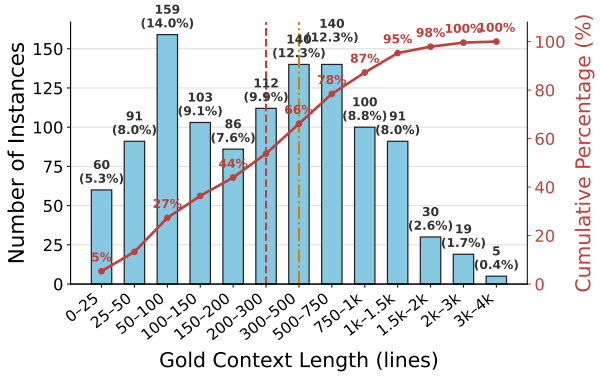


Figure 4: **Distribution of gold context length across ContextBench.** Per-instance gold contexts span a wide range of source-code line counts, reflecting the diversity of real-world repository-level tasks. The dashed red and dash-dotted orange lines denote the median (254 lines) and mean (460 lines), respectively.

E Block-Level (AST) Alignment Details

Block-Level (AST) Alignment. At the block level, we do not treat arbitrary AST nodes as evaluation units. Instead, we standardize blocks as *definition-level symbols* (e.g., function/method/class/interface/trait definitions) to ensure cross-language consistency. Concretely, we parse each repository file with `tree-sitter` and extract a language-specific set of node types that correspond to top-level or member definitions (Table 6). Each extracted node is identified by its file path, node kind, and `(start_byte, end_byte)` span. Annotated and retrieved code ranges are mapped to the definitions they overlap in this shared repository snapshot. This design ensures that "block" refers to comparable semantic units across languages (definitions that encapsulate reusable logic or APIs), rather than low-level syntax nodes whose granularity varies substantially across parsers.

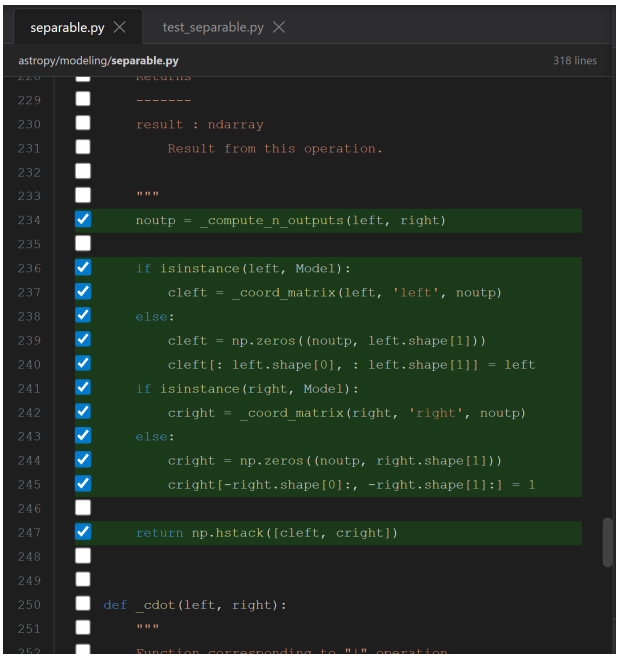
Table 6: **Tree-sitter node types used for block-level (definition-level) evaluation across languages.** Blocks are standardized as definition-level symbols to improve cross-language comparability. The table follows the extractor's language configuration, which also supports languages outside the eight evaluated in CONTEXTBENCH.

Language	Node types (block units)
Python	`function_definition`, `class_definition`, `async_function_definition`
JavaScript	`function_declaration`, `class_declaration`, `method_definition`, `arrow_function`
TypeScript	`function_declaration`, `class_declaration`, `method_definition`, `interface_declaration`
TSX	`function_declaration`, `class_declaration`, `method_definition`, `interface_declaration`
Java	`method_declaration`, `class_declaration`, `interface_declaration`, `constructor_declaration`
Go	`function_declaration`, `method_declaration`, `type_declaration`
Rust	`function_item`, `impl_item`, `struct_item`, `trait_item`
C	`function_definition`, `struct_specifier`
C++	`function_definition`, `class_specifier`, `struct_specifier`
C#	`method_declaration`, `class_declaration`, `interface_declaration`
PHP	`function_definition`, `method_declaration`, `class_declaration`
Ruby	`method`, `class`, `module`
Swift	`function_declaration`, `class_declaration`, `protocol_declaration`
Kotlin	`function_declaration`, `class_declaration`
Scala	`function_definition`, `class_definition`, `trait_definition`

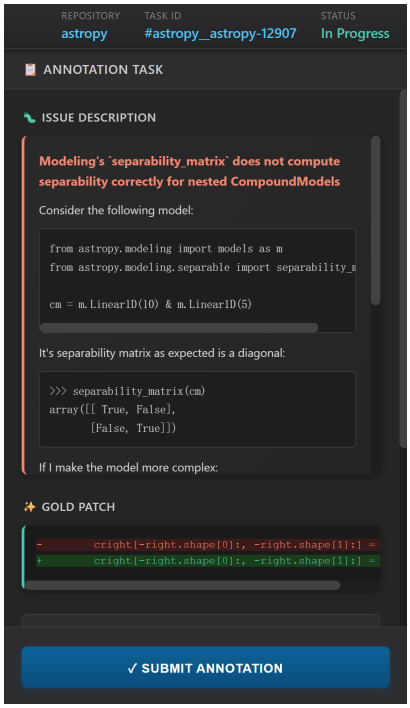
F Details on Annotation

To obtain precise problem-specific contextual annotations, we developed a dedicated annotation frontend that supports multiple annotators working in parallel with randomized task assignment. The interface provides lightweight IDE-like functionalities, including navigation via function and variable references, as well as object-oriented class definition lookup, enabling annotators to efficiently explore repository-level codebases.

Figure 5 presents the annotation frontend UI. The code annotation view (left) allows annotators to select minimal problem-specific code spans with lightweight IDE-style navigation. The side panel (right) displays structured task metadata, including issue descriptions, reference patches, and real-time annotation statistics, enabling annotators to maintain awareness of the overall task scope.



(a) Code annotation view.



(b) Sidebar with issue description and statistics.

Figure 5: Annotation frontend UI. Annotators select minimal problem-specific code context while monitoring task-level metadata in the side panel.

The Jaccard check is used only when executable verification fails. Two annotators work independently and are blinded to each other's annotations. After both annotations are complete and before joint discussion, we compute their Jaccard similarity at the line level as the intersection over union of the annotated source lines (Equation 12). If the score is at least 0.8 and the experts reach consensus, we retain the consensus context. If the score is below 0.8, a third expert joins the discussion to adjudicate differences. Contexts that pass executable verification undergo compactness checking and joint review without this Jaccard requirement. In both branches, the refined consensus context is verified again.

We refer readers to the main paper for the full pipeline, which combines independent blind annotation, joint review, and verification of refined contexts.

Our annotator team consists of highly trained software engineering experts with advanced research experience in large-scale codebases.

All annotators underwent a unified training process before participating in the annotation tasks. During training, we enforced a standardized annotation guideline, requiring annotators to provide

precise and minimal contextual information. Specifically, annotators were instructed to follow a *minimal context principle*, selecting only the code segments that were strictly necessary for resolving the issue, and avoiding redundant or irrelevant code regions.

The annotation time per issue ranged from approximately 20 minutes to 1.5 hours, depending on the complexity of the repository and the issue scope. On average, producing a high-quality annotation for a single issue in a repository-scale project requires approximately 40 minutes of effort by an expert annotator.

For repositories with exceptionally large or complex codebases, we imposed an upper bound on annotation time to control annotation cost and prevent diminishing returns from excessively long inspection sessions.

We periodically audited annotations to ensure guideline compliance and consistency across annotators.

G Details on Context Verification

This subsection provides additional details on the context verification procedure used in our analysis. The goal of this process is to determine whether an annotated context contains sufficient information to resolve its corresponding issue, rather than to evaluate or compare the repair capabilities of specific language models.

Verification Setup. For each task, we condition GPT-5 solely on the annotated context, without access to any external files, repositories, or retrieval mechanisms. The model is prompted to generate multiple independent candidate patches (five attempts in our experiments), each following the same prompt template. All generated patches are evaluated using the official test suite associated with the task, including both fail-to-pass and pass-to-pass test cases. After annotators refine a context and reach consensus, we apply this verification procedure again to the updated version. Expert review can retain a context when this final execution check fails, following the agreement and adjudication procedure in Section 2.3.

Executable Verification Criterion. Executable verification succeeds if at least one of the generated patches passes the entire test suite. This criterion is intentionally defined in terms of existence rather than consistency: we do not require the model to reliably solve the task across all attempts. Instead, we only check whether a valid solution is possible given the provided context. This aligns with our objective of verifying context sufficiency, rather than measuring average model performance.

Rationale. The check asks whether a test-passing patch can be produced from the annotated context under a fixed verifier and prompt. A single verifier is sufficient to provide a positive feasibility witness, because acceptance depends on the executable tests rather than on agreement among model judgments. It does not certify model-independent sufficiency or context minimality. Failure across five attempts remains inconclusive: it can reflect reasoning limitations or sampling variability even when the context is sufficient. Such cases undergo the additional annotation and consensus review described in Section 2.3.

Sampling and Stochasticity. We employ multiple independent generations to mitigate the effects of stochastic decoding and occasional generation failures. This design reduces false negatives caused by randomness, while avoiding over-reliance on a single generation outcome. Importantly, correctness is determined exclusively by the test suite, independent of the model's internal confidence or reasoning traces.

Overall, this verification procedure should be interpreted as a feasibility-oriented validation of annotated contexts, rather than a benchmark of language model repair performance.

H Details on Evaluation Metrics

We evaluate coding agents by quantifying how effectively they retrieve relevant context compared to the human-verified gold contexts. Formally, given an issue task T, let C^G denote the gold context, C^A denote the final declared patch-required context, and C_i^A denote the source-code regions observed

in the i-th tool interaction, recovered by parsing the command and its returned code content. All C^G, C^A, and C_i^A are represented as sets of code elements at a specified granularity (*e.g.*, file, AST block, or line). The patch-context R/P/F1 columns in the main tables compare C^A with C^G at all three granularities; the separate Explored F1 column is defined in Appendix I. The efficiency, redundancy, and drop values in Table 5 use source-code lines as their units. All evaluated trajectories contained more than one observation step, and the context sets used as metric denominators were nonempty.

Issue Resolution (Pass@1). For N evaluated tasks, let s_i equal one if the patch from a single issue-resolution run passes task i's official test suite, and zero otherwise. We define

$$\text{Pass@1} = \frac{1}{N}\sum_{i=1}^{N} s_i. \tag{1}$$

Each task is assessed with its source benchmark's test harness. The reported scores concern the selected CONTEXTBENCH tasks, rather than the full source-benchmark leaderboards. Pass@1 measures final task success, whereas context F1 measures overlap with the annotated reference; neither score determines the other. The five candidate patches used during gold-context verification serve a separate feasibility check and are not counted as five trials in Pass@1.

Context Recall. This metric quantifies the proportion of gold context included in the evaluated context set, formulated as:

$$\text{Recall}(C^A, C^G) = \frac{|C^A \cap C^G|}{|C^G|}. \tag{2}$$

A higher recall indicates greater coverage of the gold reference. A lower patch-context recall indicates that the declaration omits more gold context; it does not establish that the agent never observed that code.

Context Precision. This metric quantifies the proportion of the agent's retrieved context that overlaps with the gold context, defined as:

$$\text{Precision}(C^A, C^G) = \frac{|C^A \cap C^G|}{|C^A|}. \tag{3}$$

A higher precision indicates closer agreement with the gold reference; a lower score indicates more retrieved code outside that reference, which may still assist reasoning.

Context F1. Context recall and precision capture two complementary aspects of an agent's context retrieval behavior. High recall with low precision indicates broad coverage with substantial code outside the reference, while high precision with low recall indicates a focused set that omits some gold context. To provide a balanced evaluation, we compute the F1 score as the harmonic mean of precision and recall, defined as:

$$F_1 = \frac{2 \times \text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}. \tag{4}$$

Context Retrieval Efficiency. Beyond the final retrieved context C^A, we evaluate retrieval as a *process* along the trajectory. For each run, we log every file-observation action and construct a sequence of intermediate context snapshots $\{C_t^A\}_{t=1}^{T}$, where C_t^A is the set of code elements (at a chosen granularity) inspected at step t. We define the *cumulative explored context* up to step t as

$$A^{(t)} = \bigcup_{i=1}^{t} C_i^A. \tag{5}$$

Let $R_t = \text{Recall}(A^{(t)}, C^G)$, with $R_0 = 0$, and $\Delta R_t = R_t - R_{t-1}$. We define retrieval efficiency as

$$\text{Efficiency} = \frac{1}{T}\sum_{t=1}^{T} R_t \, \mathbf{1}[\Delta R_t > 0]. \tag{6}$$

Here $\mathbf{1}[\cdot]$ is the indicator function and T counts all observation steps in the run. A step contributes its cumulative recall only when it increases gold coverage; zero-gain steps remain in the denominator.

Penalizing observations that add no new gold evidence is an explicit design objective. Such observations consume retrieval steps without expanding the reference coverage, so they receive no credit merely for maintaining a previously attained recall. The R_t factor weights each coverage-increasing step by the cumulative gold coverage attained at that point.

Equivalently, let $K = \sum_{t=1}^{T} \mathbf{1}[\Delta R_t > 0]$ be the number of observations that increase gold coverage. For $K > 0$, the score can be written as

$$\text{Efficiency} = \frac{K}{T}\left(\frac{1}{K}\sum_{t:\,\Delta R_t>0} R_t\right).$$

It therefore combines the fraction of observations that acquire new gold evidence with the average cumulative recall at those observations. If $K = 0$, the original definition gives zero. A trajectory with higher cumulative recall can consequently receive a lower score if more of its observation steps yield no additional gold coverage; this ranking reflects the intended penalty for unproductive retrieval steps. Here, productivity is defined specifically with respect to acquiring new gold context: a zero-gain observation may still assist reasoning or verification. We scope this metric to productive gold-context acquisition per observation, without interpreting it as intermediate coverage timing or overall problem-solving efficiency.

Context Retrieval Redundancy. Efficiency credits steps that increase gold coverage, while redundancy measures repeated inspection of previously observed code. We therefore quantify redundancy by measuring how much of each newly retrieved snapshot overlaps with what has already been observed. Formally, for $t \geq 2$, we define the *per-step redundancy ratio* as the fraction of elements in C_t^A that have appeared in previous steps:

$$\text{Redun}_t \;=\; \frac{\left|C_t^A \cap \left(\bigcup_{i=1}^{t-1} C_i^A\right)\right|}{\left|C_t^A\right|}. \tag{7}$$

We report the overall redundancy as the average across steps:

$$\text{Redun} \;=\; \frac{1}{T-1}\sum_{t=2}^{T}\text{Redun}_t. \tag{8}$$

A higher redundancy indicates that an agent spends more retrieval budget revisiting previously seen context (i.e., looping or re-reading), whereas a lower redundancy indicates more *novel* context acquisition per step.

Evidence Drop from Exploration to Patch Context. Evidence drop compares gold coverage in the final declared patch context C^A with gold coverage observed during exploration. When the declared gold lines are a subset of the observed gold lines, it measures the fraction of observed gold evidence absent from the declaration (Appendix I). To quantify this gap, we define the *explored-gold set* as the gold elements that have been observed at least once during the trajectory:

$$G_{\text{seen}} \;=\; \left(\bigcup_{t=1}^{T} C_t^A\right) \cap C^G. \tag{9}$$

We define the declared-to-explored gold-coverage ratio as

$$\text{Keep} \;=\; \frac{\left|C^A \cap C^G\right|}{|G_{\text{seen}}|}, \tag{10}$$

and define *evidence drop* as its complement:

$$\text{Drop} \;=\; 1 - \text{Keep} \;=\; 1 - \frac{\left|C^A \cap C^G\right|}{\left|\left(\bigcup_{t=1}^{T} C_t^A\right) \cap C^G\right|}. \tag{11}$$

A lower Drop (higher Keep) indicates higher declared gold coverage relative to explored gold coverage. Its interpretation as a fraction of observed evidence omitted assumes the containment condition above.

Gold Context Robustness. Our gold contexts are human-annotated based on a gold patch. However, in practice, a single issue may admit multiple semantically equivalent but syntactically different patches, raising the question of whether the collected gold context is reliable under solution multiplicity. Alternative patches are collected from our own agent runs and publicly released experimental results associated with the source benchmarks' leaderboards. The analysis includes 331 tasks from CONTEXTBENCH with at least two distinct test-passing patches available per task, yielding 884 patches in total. Task inclusion is determined by patch availability. We derive a patch-conditioned gold context for each patch using the same context tracing procedure. Let $\mathcal{G} = \{G^{(1)}, \ldots, G^{(K)}\}$ denote the set of gold contexts for a given instance. We quantify consistency using the Jaccard similarity (Equation 12) between each pair of contexts. For each instance, we report the average similarity across all context pairs.

$$\text{Jaccard}(G^{(i)}, G^{(j)}) = \frac{|G^{(i)} \cap G^{(j)}|}{|G^{(i)} \cup G^{(j)}|}. \tag{12}$$

At the block level, every evaluated context pair has Jaccard similarity exactly equal to 1, before rounding: the patch-conditioned contexts for each task contain identical block sets. The results characterize context consistency among the collected patches on this subset.

All reported metrics are macro-averaged over evaluated tasks: recall, precision, and F1 are computed separately per task and then averaged, so mean F1 need not equal the harmonic mean of mean recall and mean precision. Recall, precision, F1, and context similarity can be computed at the file, block, and line levels; the reported process metrics use the line level. By evaluating context retrieval across multiple metrics and granularities, CONTEXTBENCH provides a more fine-grained understanding of agent behavior, beyond conventional end-to-end evaluation metrics that focus solely on final task success rates.

I EXPLORED CONTEXT AND FINAL PATCH-CONTEXT SELECTION

I.1 DEFINITIONS AND MEASUREMENT

We distinguish three targets: source code exposed to an agent during execution, source code the agent ultimately declares relevant to its patch, and information that causally influences patch generation. The first two are observable through trajectory records and final declarations, respectively. Neither establishes the third.

For task i, let $C^A_{i,t}$ contain the implementation source-code regions actually returned to the agent by tools at observation step t, and let T_i be the number of such steps in the complete run. Its *explored context* is the cumulative, deduplicated union

$$E_i = \bigcup_{t=1}^{T_i} C^A_{i,t}. \tag{13}$$

Let D_i denote the final `<PATCH_CONTEXT>` declaration and G_i the gold context. The declaration lists source regions judged relevant to generating the patch, including unchanged definitions or dependencies; it is not the patch diff and is not restricted to edited lines. In the notation of Appendix H, E_i corresponds to $A^{(T_i)}$ and D_i to C^A. Explored-context scores compare E_i with G_i; patch-context scores compare D_i with G_i. These are separate measurements, even when the declaration is requested from previously inspected code.

Observation and scope. Only source content actually exposed in tool outputs contributes to E_i. A file-reading command contributes the returned lines; a search contributes the code lines it prints. Commands returning only filenames, directory entries, or status information do not establish observation of file contents. Requested ranges hidden by truncation, omitted search matches, and unreturned file contents must not be inferred from the command alone. Repeated or overlapping reads contribute each source element only once. As with the gold reference, the scored universe is repository implementation source code; tests, documentation, issue text, and execution traces are outside this comparison. All observed implementation source, including files absent from G_i, remains in E_i and contributes to its precision denominator. Excluding other artifact types defines the measurement scope and does not imply that they lack diagnostic value.

Coordinate alignment. The comparison requires explored and declared regions to use the same repository snapshot as G_i, with normalized repository-relative paths and source coordinates. Line units are pairs of file paths and line numbers. File units are the files containing observed source lines; block units are the definition-level AST nodes overlapped by these lines, following Appendix E. Observing part of a block can therefore identify that block without implying that its other lines were observed. Output truncation and coordinate recovery must be handled at the tool-interface level. In particular, source returned after an edit requires mapping back to the reference snapshot before overlap is scored; edited-file line numbers alone do not identify reference lines. Content without a reliable snapshot mapping cannot be credited as observed reference code.

Per-task scoring and aggregation. At each granularity, we apply the recall, precision, and F1 definitions in Appendix H separately to each task, using the same evaluated tasks as the corresponding patch-context comparison and the nonempty-denominator setting specified there. Each reported R, P, or F1 is then its own arithmetic mean over tasks, with equal task weights. In particular, mean F1 is not the harmonic mean of mean recall and mean precision. The main-table Explored F1 is the line-level value; no average across granularities is used. This unit matches the existing Efficiency, Redundancy, and Drop measurements and permits direct comparison with patch-context line-level F1.

I.2 Complete Explored-Context Results

Tables 7 and 8 report file-, block-, and line-level explored-context R/P/F1 for the seven agents and seven model backbones. The agent comparison follows the GPT-5 setting of Table 2; the backbone comparison follows the mini-SWE-agent setting of Table 3. The line-level F1 values are reproduced in the Explored F1 columns of those main tables, alongside the patch-context results.

For every reported agent and model at each granularity, mean explored recall is higher than mean patch recall, while mean explored precision and F1 are lower. Cumulative observation thus covers more of the reference but also includes proportionally more code outside it. At the reported precision, the F1 ordering, including ties, is preserved within each comparison at all three granularities. This agreement is an empirical result of these evaluations; the two measurement targets need not produce the same ranking in general.

At the line level, Agentless leads the agent comparison with explored F1 of 0.357, compared with patch F1 of 0.376; Claude Code and Codex follow with explored F1 of 0.311 and 0.305. In the backbone comparison, Claude Opus 4.7 leads with explored F1 of 0.340, compared with patch F1 of 0.358, followed by GPT-5.4 at 0.330 and Claude Sonnet 4.5 at 0.327. These paired measurements separate broad acquisition from the more selective final declaration without changing the reported patch-context results.

Table 7: **Explored-context results for seven coding agents using GPT-5.** R, P, and F1 compare cumulative, deduplicated observations with gold context; all are macro-averaged over tasks. Bold denotes the best value in each column. Patch-context results are in Table 2.

Coding agent	File			Block			Line		
	R↑	P↑	F1↑	R↑	P↑	F1↑	R↑	P↑	F1↑
Agentless	0.752	0.271	0.371	0.591	0.160	0.230	0.632	**0.249**	**0.357**
mini-SWE-Agent	0.872	**0.485**	0.602	0.811	0.261	0.356	0.765	0.208	0.296
SWE-agent	0.889	0.396	0.517	0.805	0.195	0.271	0.695	0.135	0.198
OpenHands	0.904	0.312	0.440	0.751	0.123	0.181	0.687	0.084	0.124
Prometheus	0.910	0.261	0.383	0.827	0.185	0.271	0.769	0.144	0.219
Codex	0.918	0.471	**0.617**	0.848	**0.267**	**0.381**	0.798	0.212	0.305
Claude Code	**0.929**	0.459	0.610	**0.861**	0.261	0.374	**0.811**	0.211	0.311

Table 8: **Explored-context results for seven model backbones using mini-SWE-agent.** R, P, and F1 use the same observation scope and aggregation as Table 7. Bold denotes the best value in each column. Patch-context results are in Table 3.

Model	File			Block			Line		
	R↑	P↑	F1↑	R↑	P↑	F1↑	R↑	P↑	F1↑
Gemini 2.5 Pro	0.801	0.464	0.570	0.714	0.277	0.383	0.660	0.190	0.295
Devstral 2	0.873	0.461	0.584	0.821	0.282	0.401	0.790	0.201	0.315
Gemini 3.1 Pro	0.838	0.459	0.590	0.727	0.301	0.407	0.660	0.207	0.313
GPT-5	0.872	0.485	0.602	0.811	0.261	0.356	0.765	0.208	0.296
Claude Sonnet 4.5	0.886	0.476	0.593	0.814	0.300	0.399	0.760	0.230	0.327
GPT-5.4	0.927	**0.591**	**0.699**	0.852	0.342	0.438	0.775	0.238	0.330
Claude Opus 4.7	**0.950**	0.590	0.692	**0.873**	**0.359**	**0.458**	**0.795**	**0.243**	**0.340**

I.3 Relationship to Retrieval Dynamics

Explored-context R/P/F1 summarizes the union at the end of a run; Efficiency and Redundancy describe how it was acquired. Efficiency credits observation steps that add gold coverage and accounts for all observation steps in its denominator. Redundancy measures repeated inspection. A repeated read can increase redundancy without increasing the deduplicated explored set or its recall. These metrics therefore describe different properties of retrieval, as formalized in Appendix H.

For a fixed snapshot and gold reference, cumulative explored recall cannot decrease as observations accumulate. Precision may decrease when new observations add proportionally more code outside the reference, and F1 depends on the changes in both recall and precision. Comparing explored context with the final declaration requires an additional condition: explored recall is at least patch recall when every declared gold line was observed. The declaration protocol requests selection from inspected code, but that request alone does not verify this containment for each trajectory.

For task i with positive explored line recall, the existing Drop definition yields

$$\mathrm{Drop}_i = 1 - \frac{R^{\mathrm{patch}}_{i,\mathrm{line}}}{R^{\mathrm{explored}}_{i,\mathrm{line}}}. \tag{14}$$

This is a *per-task* identity. Mean Drop and mean patch recall cannot be substituted into it to recover mean explored recall, because averaging does not commute with the ratio. If no gold line was observed, this ratio is undefined and its handling must be reported explicitly. Interpreting Drop as the fraction of observed gold evidence absent from the declaration further requires $D_i \cap G_i \subseteq E_i \cap G_i$. Without that condition, the formula describes a relative recall gap, which can be negative, rather than a retention fraction. Even with containment, Drop measures a gap between observation and selection; it does not demonstrate forgetting or causal non-use.

I.4 Architectural Differences and Interpretation

Why request a final declaration? In a staged pipeline such as Agentless, localization and context selection precede a comparatively separate patch-generation stage. The full exploration history may contain code outside the input selected for that stage. In iterative agents, retrieval, reasoning, editing, and testing share an interaction history. Imposing an equivalent boundary around the information available for patch generation would change that workflow. A pre-submission declaration makes the agent's own relevance selection explicit and provides a common operational target across these architectures; cumulative explored context separately records the source code exposed during the run.

What the two measurements establish. A final declaration is a reported selection, not a reconstruction of the model's internal information use. Earlier observations may influence later reasoning even when their source regions are omitted from the declaration. Conversely, inclusion in a declaration, tool output, or pipeline component's input does not establish that the region affected the patch. Recovering tool observations does not require separating patch generation from earlier interactions, but causal attribution would require evidence beyond these records. Patch-context F1, explored-context F1, and Pass@1 consequently answer different questions: agreement of final

selection with the reference, agreement of cumulative observations with the reference, and successful issue resolution.

How to interpret precision. The gold reference is a compact, human-verified implementation context, with no guarantee of global minimality (Appendix D). Additional implementation code may help diagnose an issue, rule out alternatives, or verify a change despite falling outside that reference. Low explored precision therefore indicates broad observation relative to the reference; it does not establish that every non-gold line was useless. Artifacts outside the implementation-code scope can also be useful, but are not included in either scored context. Together, the two context measurements make acquisition and final selection visible while preserving this limit on claims about usefulness and causal influence.

J Details on Empirical Study

This subsection provides implementation-level details of our empirical analysis. To analyze agent behavior beyond final patch correctness, we instrument the evaluation pipeline to record intermediate artifacts, including tool-call logs, retrieved file and element contexts, and step-by-step edit trajectories.

Based on these records, we perform manual inspection to trace how each agent localizes relevant files, retrieves semantic information, and arrives at its final fix. All annotation and analysis are conducted with the support of the annotation frontend UI introduced earlier, which enables structured inspection and labeling of the recorded artifacts. The following case studies illustrate this analysis procedure for different agents, highlighting how the recorded artifacts are used to diagnose failure points at various stages of the debugging process.

J.1 Case 1: Prometheus Agent (Incomplete Class Semantics Retrieval)

Task: Fix issue `psf__requests-1921` where setting session headers to `None` sends literal "None" instead of omitting the header.

Diagnostic Findings:

1. **Partial Class Retrieval**: Agent logs show successful retrieval of `CaseInsensitiveDict`'s `__setitem__` and `__getitem__` methods (lines 71-78), but *no retrieval* of its `__init__` or `update` methods.
2. **Semantic Mismatch**: The generated patch passed a generator expression to `CaseInsensitiveDict` (line 54 of `requests/sessions.py`):

   ```
   merged_setting = dict_class((k, v) for k, v in ... if v is
       not None)
   ```

   Analysis of the missing `__init__` reveals it requires a mapping with `.items()`, not a generator.
3. **Impact**: 26 test failures due to dropped headers, broken authentication, and cookie persistence failures.

Gold Patch Comparison: The correct fix adds a single filtering line *after* merging, preserving proper initialization:

```
merged_setting = dict((k, v) for (k, v) in merged_setting.items()
    if v is not None)
```

Root Cause: **Element-level slicing insufficiency**. The agent retrieved operational methods but missed constructor semantics, leading to API contract violation.

J.2 Case 2: Agentless (Failure at File Localization Stage)

Task: Fix for `django__django-11630`.

Diagnostic Findings:

1. **Stage 1 Failure (File Localization)**: The agent retrieved 10 files, including 5 from `django/db/models/*` (e.g., `base.py`, `options.py`, `fields/`) and 5 framework configuration files. Critically, `django/core/checks/model_checks.py` (the source of error `E028` based on gold patch analysis) was **never retrieved**.

2. **Cascading Failure**: With incorrect files localized in Stage 1, subsequent element localization (Stage 2) focused on `ModelBase`, `Options` classes, and edit localization (Stage 3) modified `options.py`'s `db_table` calculation. The gold patch, by contrast, modifies logic in `model_checks.py` to conditionally emit warnings when database routers are configured.

3. **Information-Architecture Gap**: The issue mentions "`db_table` collision" (symptom-level) but not the validation framework (implementation-level). File tree shows `core/checks/` exists but lacks semantic annotations. Without backward-tracing from error code `E028` to its source, the agent inferred the problem resided in model definition layers.

Root Cause: **File localization failure propagated through all stages**. Operating on surface-level keywords from the issue ("models", "db_table"), the agent lacked mechanisms to: (i) trace error codes to implementation sources, (ii) recognize Django's architectural separation of validation from model logic. This initial misstep rendered all downstream localization (element and edit) ineffective, as they operated on an incorrect file set.

J.3 Case 3: OpenHands Agent (Cross-Context Exploration Gap)

Task: Fix for `django__django-11138` regarding ignored `TIME_ZONE` values in multi-backend configurations.

Diagnostic Findings:

1. **Keyword-Centric Search Bias**: The agent relied heavily on `grep` for string matching. Since this specific SQL function is unique to MySQL, the tool's output anchored the agent's search space to the MySQL implementation, effectively shielding the relevant SQLite and Oracle modules from the localization process.

2. **Structural Observation-Action Gap**: While `ls -R` tool calls explicitly listed the `sqlite3/` and `oracle/` directories, the agent's heuristic failed to trigger follow-up audit calls (e.g., `view` or targeted `grep`) for these parallel modules. This represents a failure to translate structural awareness into diagnostic action.

3. **Cross-Backend Reproduction Paradox**: Faced with environment constraints for MySQL, the agent used `bash` tool calls to successfully run a reproduction script in a SQLite environment. Although the tool-verified evidence confirmed the bug's presence in SQLite, the localization logic failed to propagate this feedback, resulting in a "reproduce in SQLite, fix in MySQL" logical disconnect.

4. **Canonical Pattern Neglect**: Despite retrieving the base operations class via `view`, the agent bypassed the framework's canonical `connection.timezone_name` property. Instead, it utilized tool calls to inject non-standard dictionary-access logic (`settings_dict.get('TIME_ZONE')`), failing to align with Django's architectural standards.

Root Cause: **Search-Induced Context Tunneling**. The agent's localization strategy is highly sensitive to initial tool-call results but lacks **semantic generalization**. High-relevance hits in one module (MySQL) acted as a distractor, suppressing the horizontal exploration of parallel modules (SQLite/Oracle). This reliance on keyword-driven slicing leads to systemic fix incompleteness in modular architectures where logic is distributed across similar but syntactically distinct files.

K RADAR PLOTS FOR CONTEXT RETRIEVAL PERFORMANCE

Overall Agent and LLM Comparisons

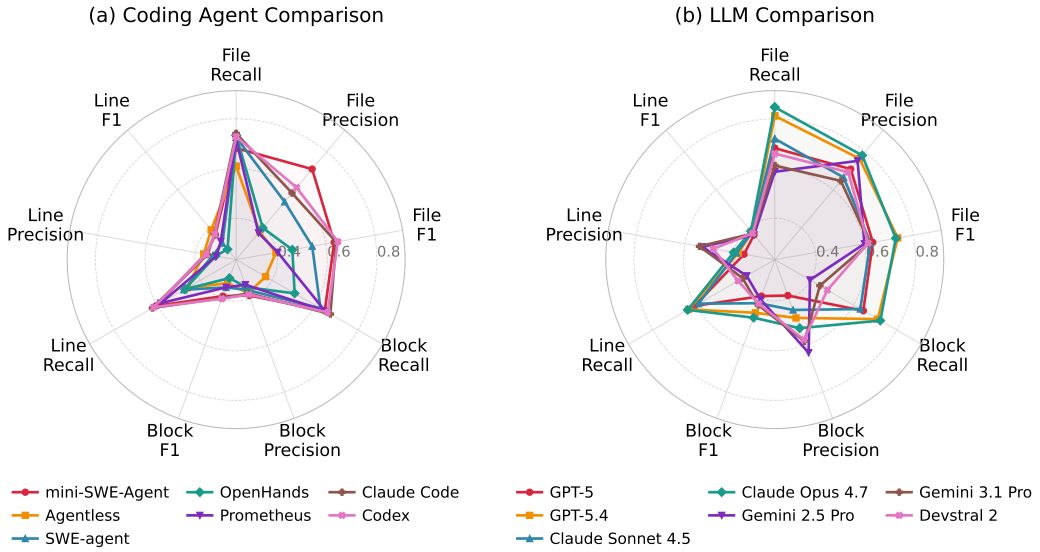
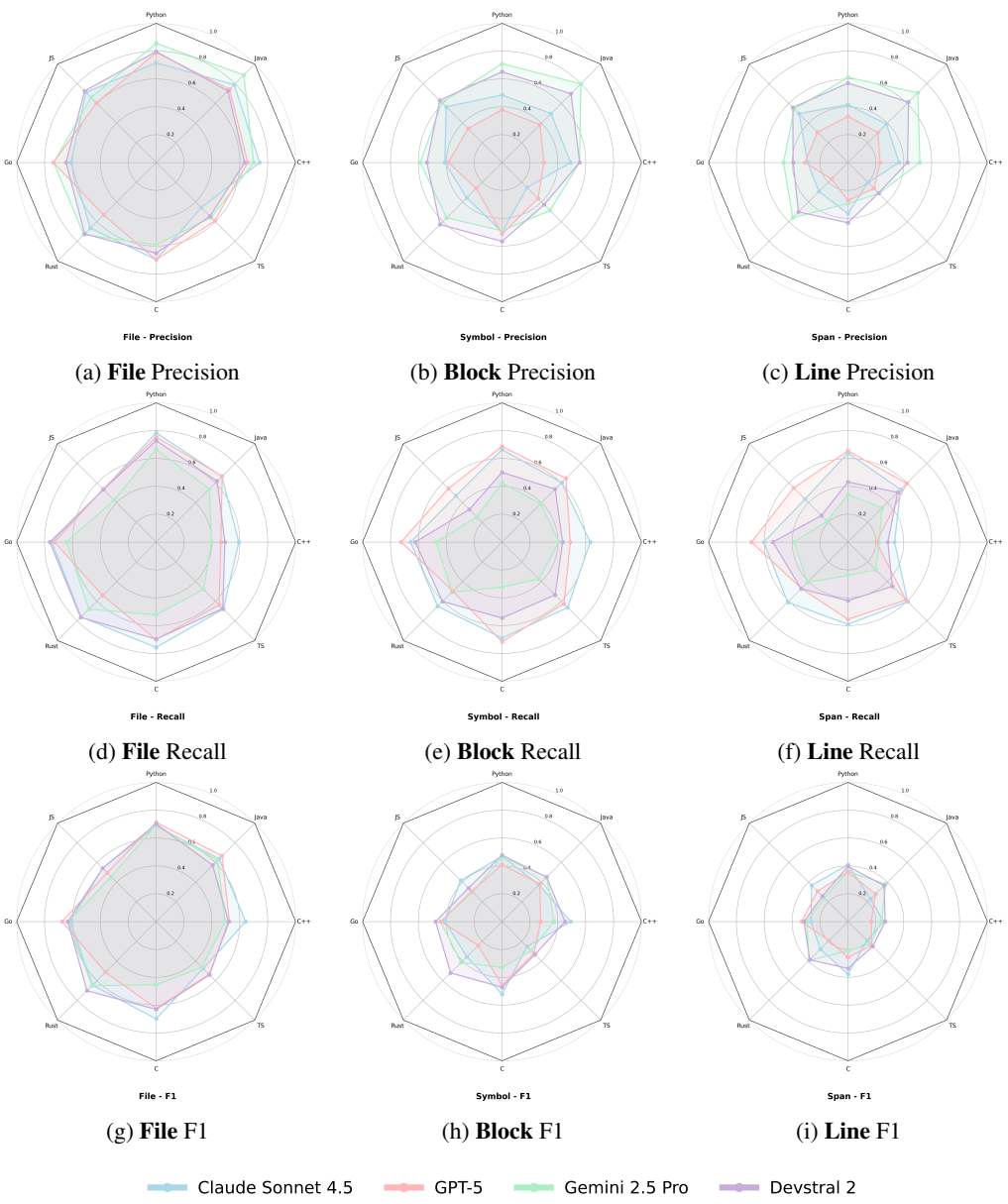


Figure 6: **Radar plots comparing declared patch-context retrieval on CONTEXTBENCH.** (a) Seven coding agents and (b) seven LLMs, evaluated at the file, block, and line levels in terms of precision, recall, and F1. Most evaluated LLMs and agents show higher recall than precision at the block and line levels. For readability, the radial axis uses a piecewise scale that compresses values from 0 to 0.4 and expands values from 0.4 to 0.9; tick labels report the original metric values.



(a) **File** Precision (b) **Block** Precision (c) **Line** Precision

(d) **File** Recall (e) **Block** Recall (f) **Line** Recall

(g) **File** F1 (h) **Block** F1 (i) **Line** F1

Figure 7: Radar distributions of declared patch-context localization across eight programming languages for GPT-5, Claude Sonnet 4.5, Gemini 2.5 Pro, and Devstral 2. Columns correspond to granularity (**File / Block / Line**), and rows correspond to metric (**Precision / Recall / F1**). Each radar axis is one language; values are computed on the per-language split under the same evaluation protocol as the main paper.

L LIMITATIONS

While CONTEXTBENCH provides a process-level lens on context retrieval, several limitations remain.

(i) Context observability and reference scope. Observed source code and final patch-context declarations provide complementary measurements, but neither establishes which evidence causally influenced a patch. Evidence drop therefore measures a gap between observation and selection rather than demonstrated forgetting. Our gold reference covers implementation source code, so

precision reflects agreement with that reference rather than the usefulness of every inspected artifact. Appendix I provides the full definitions, results, and architectural discussion.

(ii) Scope of executable verification. We use a single verifier, GPT 5, with five attempts. Expert consensus or adjudication can support acceptance when executable verification fails. A successful execution check demonstrates feasibility under this verifier and prompt; other verifiers may succeed on different contexts.

(iii) Patch and language coverage. While we span eight programming languages and 66 repositories drawn from four source benchmarks, the distribution still skews toward Python; results on under-represented languages and on private/proprietary codebases may not generalize.

(iv) Agent and model coverage. We benchmark seven widely used coding agents and seven frontier LLMs, but rapidly evolving systems, particularly long-context, agentic-tool, or memory-augmented variants, are not exhaustively covered, and absolute numbers may shift with newer releases.

(v) Snapshot evaluation. Each task is evaluated against a fixed repository snapshot; this isolates retrieval behavior but does not capture multi-revision or live-repository dynamics where context drifts over time.

M Evaluating Context Retrieval as an Agent Stage

A stage-oriented evaluation perspective. ContextBench introduces an explicit evaluation of context retrieval within the issue-resolution process. It measures the source-code evidence an agent acquires and selects, making this intermediate stage observable alongside final patch correctness. RQ1 asks how effectively different agents perform this stage. The unit of comparison is each agent's complete context retrieval mechanism: its tools, prompts, search strategy, context management, and stopping decisions jointly determine the context it retrieves. Preserving these mechanisms allows the benchmark to compare how different agent designs perform the same functional stage.

Shared evaluation conditions. In Table 2, agents share the GPT-5 backbone, evaluated tasks, per-task repository snapshots, gold contexts, and scoring definitions at the file, block, and line levels. Patch-context and explored-context scores measure overlap with the same annotated reference and are averaged over tasks, as specified in Appendix H. Pass@1 separately measures patch correctness. These shared conditions establish a common basis for evaluating retrieved content while retaining the agents' own retrieval interfaces and interaction strategies. RQ2 addresses a complementary question by fixing mini-SWE-agent and varying the model backbone.

Retrieval within the repair process. Context retrieval can recur throughout a repair run as the agent plans, edits, and tests its solution. Execution feedback can motivate further searches, and newly retrieved evidence can inform subsequent edits. ContextBench records context acquisition across these interactions and the context selected for the final patch. This tracing protocol evaluates retrieval as a functional stage within the agent's complete workflow, preserving the way each context mechanism operates during issue resolution.

What the stage-level comparison reveals. For declared patch context, the three shell-based agents occupy the top three positions in file-level F1 (0.634 to 0.649) and block-level F1 (0.375 to 0.401). Claude Code and Codex also achieve the highest Pass@1 scores, 0.534 and 0.522, respectively. At the line level, Agentless obtains the highest F1 (0.376), compared with 0.312 for mini-SWE-agent, 0.327 for Claude Code, and 0.321 for Codex. The retrieval metrics also reveal differences beyond the final success ranking: mini-SWE-agent scores above Prometheus in F1 at the file and block levels, whereas their Pass@1 scores are 0.472 and 0.512, respectively. These distinct profiles demonstrate the value of evaluating context retrieval directly alongside task completion.

Implications for context mechanism design. Together, these measurements support RQ1's design finding: simple shell-based context mechanisms achieve strong retrieval performance, and added retrieval machinery does not consistently improve context quality across the evaluated agents. The discussion of over-engineering and the "Bitter Lesson" concerns these complete mechanisms at the context retrieval stage. By exposing their intermediate retrieval outcomes, ContextBench makes context acquisition and selection explicit dimensions for comparing agent designs in repository-level problem solving.

N OBSERVED INTERACTION AND PROTOCOL MISALIGNMENT IN DEVSTRAL 2

During our evaluation with the mini-SWE-agent framework, we observed a recurring interaction pattern for Devstral 2. After reading the issue description, the model sometimes produced a long, structured solution plan before receiving environment feedback and occasionally referred to intermediate states that had not yet been returned by the framework. When the framework subsequently reported an invalid output format, the model sometimes proceeded directly to the final `submit` action rather than performing additional corrective iterations.

These observations are specific to the evaluated combination of model, agent, and prompt and should not be interpreted as evidence of training-data contamination or as a general deficiency of Devstral 2. Plausible explanations include a mismatch between the interaction protocol learned by the model and mini-SWE-agent's one-action-per-turn interface, strong priors about common coding-agent trajectories, or sensitivity to the framework's output-format constraints.

We therefore report this behavior as a protocol-compatibility case study rather than making a causal claim about the model or its training data. Future evaluations should document model-specific adapters, validate action-format compliance, and separate protocol failures from context-retrieval quality when comparing coding agents.

DevStral2 Raw JSON Response Trace

Listing 1: DevStral2 Raw JSON Response Trace


```
"extra": {
    "response": {
        "id": "72454b08842c433ab3c5e23b0b8e4394",
        "created": 1768642675,
        "model": "devstral-2512",
        "object": "chat.completion",
        "system_fingerprint": null,
        "choices": [
        {
            "finish_reason": "tool_calls",
            "index": 0,
            "message": {
            "content": "I'll start by exploring the Django
   codebase...
            <EXPLORE_CONTEXT>\n
            File: /testbed/django/core/management/base.py\
   nLines: 1-100\n
            </EXPLORE_CONTEXT>\n\n
            ```bash\nfind /testbed -type f -name \"base.py\"
   | \
            grep \"management\" | head -5\n```\n\n
            ...
            ...
            Perfect! I found the issue.
            Now let me create a test script to reproduce the
   problem...
            ```bash\ncat <<'EOF' > /testbed/
   test_mutually_exclusive.py\nimport os\nimport sys\n
            import django\nfrom django.core.management import
    call_command\n
            from django.core.management.base import
   BaseCommand, CommandError\n
            ...
            Let me run the core management tests:\n\n
```

```
            ```bash\ncd /testbed && python -m pytest tests/
    core_management/ -v 2>&1 | head -100\n```\n\n
            Let me check what tests exist:\n\n
            ```bash\nfind /testbed/tests -type f -name \"*.py
    \" | xargs grep -l \"mutually_exclusive_group\" | head -5
    \n```\n\n
            Let me check if there's a test file for the
    management module:\n\n
            ```bash\nfind /testbed/tests -type f -name \"*
    management*.py\" | head -10\n```\n\n
            ...
            Now let me verify the fix one more time and check
     if there are any syntax errors...
            Great! The tests pass. Let me create a final
    summary of the changes...
            The fix is minimal, handles edge cases, and doesn
    't break existing functionality.",
            "role": "assistant",
            "tool_calls": null,
            "function_call": null,
            "provider_specific_fields": null
            },
            "provider_specific_fields": {}
        }
        ],
        "usage": {
        "completion_tokens": 13989,
        "prompt_tokens": 1836,
        "total_tokens": 15825,
        "completion_tokens_details": null,
        "prompt_tokens_details": null
        }
    }
}
{
    "role": "user",
    "content": "Please always provide EXACTLY ONE action in
    triple backticks, found 107 actions.\n\nPlease format
    your action in triple backticks as shown in <
    response_example>.\n\n<response_example>\nHere are some
    thoughts about why you want to perform the action.\n\n```
    bash\n<action>\n```\n</response_example>\n\nIf you have
    completed your assignment, please consult the first
    message about how to\nsubmit your solution (you will not
    be able to continue working on this task after that).",
    "timestamp": 1768642877.7618906
},
"extra": {
        "response": {
          "id": "486b79cc024b4735a91b0a1a8ca7c909",
          "created": 1768642878,
          "model": "devstral-2512",
          "object": "chat.completion",
          "system_fingerprint": null,
          "choices": [
            {
              "finish_reason": "stop",
```
```

```
            "index": 0,
            "message": {
              "content": "THOUGHT: I have successfully
    implemented the fix for the issue where `call_command`
    fails when an argument of a required mutually exclusive
    group is passed in kwargs. The fix has been applied to `/
    testbed/django/core/management/__init__.py` and
    thoroughly tested. Now I need to submit the solution.\n\n
    ```bash\necho COMPLETE_TASK_AND_SUBMIT_FINAL_OUTPUT && (
    git status >/dev/null 2>&1 && git add -A && git diff --
    cached) || (cd */. 2>/dev/null && git add -A && git diff
    --cached) || (echo \"Error: No git repository found\")\n
    ```",
              "role": "assistant",
              "tool_calls": null,
              "function_call": null,
              "provider_specific_fields": null
            },
            "provider_specific_fields": {}
          }
        ],
        "usage": {
          "completion_tokens": 136,
          "prompt_tokens": 15925,
          "total_tokens": 16061,
          "completion_tokens_details": null,
          "prompt_tokens_details": null
        }
      }
    }
```