arXiv:2602.11988v3 [cs.SE] 29 Sep 2026

Evaluating AGENTS.md: Are Repository-Level Context Files Helpful for Coding Agents?

Thibaud Gloaguen
Department of Computer Science
ETH Zurich
thibaud.gloaguen@inf.ethz.ch

Niels Mündler-Sasahara
Department of Computer Science
ETH Zurich
niels.muendler@inf.ethz.ch

Mark Niklas Müller
LogicStar.ai

Veselin Raychev
LogicStar.ai

Martin Vechev
Department of Computer Science
ETH Zurich

Abstract

A widespread practice in software development is to tailor coding agents to repositories using context files, such as `AGENTS.md`. Although this practice is strongly encouraged by agent developers, there is currently no rigorous investigation into whether such context files are actually effective for real-world tasks. In this work, we study this question and evaluate coding agents' task completion performance in two complementary settings: established SWE-bench tasks from popular repositories, with LLM-generated context files, and a novel collection of issues from repositories containing developer-committed context files. Surprisingly, we find that providing context files does not generally improve task success rates, while *increasing inference cost* by over 20% on average. This observation holds across different LLMs, coding agents, and for both LLM-generated and developer-committed context files. Specifically, we find that while instructions in the context files are well followed by coding agents, repository overviews, although popular and recommended by model providers, are not helpful. We conclude that while context files are useful for specifying non-standard coding practices, any attempts to improve performance should be rigorously evaluated before deployment.

1 Introduction

Coding agents are being rapidly adopted across the software engineering industry [22, 29], and providing context files like `AGENTS.md`, a `README` specifically targeting agents, has become common practice. With various industry leaders [1, 4] recommending this approach to adapt their agents to specific repositories, context files are now supported by most popular agent frameworks, and included in over 60'000 open-source repositories, as reported by AGENTS.md [1].

These context files typically contain a repository overview and information on relevant developer tooling, aiming to help coding agents to navigate a given repository more efficiently, run build and test commands correctly, adhere to style guides and design patterns, and ultimately to solve tasks to the user's satisfaction more frequently. To date, despite their widespread adoption, the impact of context files on the coding agent's ability to solve complex software engineering tasks has not been rigorously studied. This is due to two key challenges: i) because of their recent introduction, context files are not available for instances of prior benchmarks, and ii) popular, well-known repositories, typically used to create such benchmarks, are not representative of most codebases. As a result, a

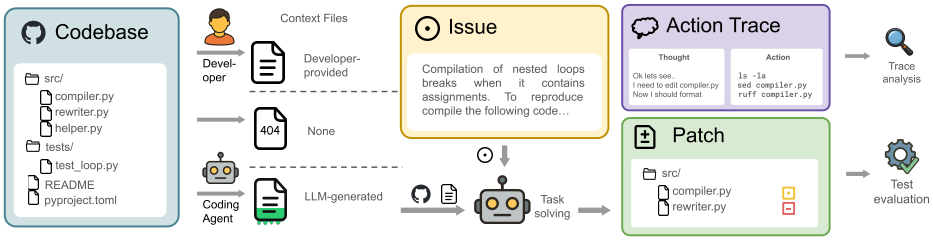


Figure 1: Overview of our evaluation pipeline. We begin with real-world repositories and tasks derived from past pull requests. For each repository state, we generate three settings: ① If a developer-provided context file exists, we include it in the repository. In ②, we omit the context file. ③ We use the coding agent's recommended settings to generate the context file. Then we pass the repository and context file to the coding agent and instruct it to resolve the task. We finally analyze the trace for behavioral changes and apply the generated patch to check for task resolution success.

rigorous evaluation of real-world context file use requires a new, complementary benchmark that contains only issues from less popular repositories with developer-committed context files.

This work: Benchmarking context files' impact on resolving GitHub issues In this work, we investigate the effect of actively used context files on the resolution of real-world coding tasks. We evaluate agents both in popular and less-known repositories, and, importantly, with context files committed by repository developers. For this purpose, we construct a novel benchmark (Figure 1, left), CTXBENCH, comprising Python software engineering tasks, created specifically from real GitHub issues. The benchmark contains 138 unique instances, covering both bug-fixing and feature addition tasks across 12 recent and niche repositories, all featuring developer-committed context files. CTXBENCH complements SWE-BENCH, which we leverage for the evaluation of automatically generated context files on popular repositories. We evaluate coding agents in three settings (Figure 1, middle): without any context file, with context files automatically generated using agent-developer recommendations, and with the developer-committed context file.

We observe that including context files does not significantly affect performance. However, *developer-committed* files outperform *LLM-generated* ones by a significant margin of 7% on average. These observations are robust across LLMs, agents, and prompts used to generate the context files. In a more detailed analysis of the agent traces, we observe that instructions in context files are well followed. This leads to increased exploration, testing, and reasoning by coding agents, and, as a result, increases costs by over 20%. We therefore suggest omitting LLM-generated context files for the time being, contrary to agent developers' recommendations, as they don't help improve performance. Human-written context files should only include instructions required for coding agents that are not already present in the `README` (e.g., specific conventions or non-functional requirements), and be rigorously evaluated before adoption. We hope our evaluation framework will aid agent and model developers to improve the helpfulness of LLM-generated context files.

Key contributions Our key contributions are:

1. CTXBENCH, a new benchmark for the impact of actively used context files on agents' ability to solve real-world software engineering tasks.
2. An extensive evaluation of different coding agents on CTXBENCH and SWE-BENCH, showing that context files don't significantly improve performance.
3. A detailed investigation of agent traces, showing that context files instructions are well followed, leading to more thorough testing and exploration by coding agents.

2 Background and Related Work

Coding agents Coding agents are LLM-based systems designed for autonomous resolution of coding tasks [39]. Typically, they consist of a harness that allows an LLM to interact with its environment using specialized tools for, e.g., executing bash commands, conducting web searches, or reading, creating, or modifying files [38, 39].

Their impressive performance on repository-level coding tasks like SWE-bench [15] led to rapid adoption in the software engineering community [29] and the development of new agents by specialized companies [2, 38] and model providers [3, 13, 26, 28]. Model providers now train their LLMs to use the tools exposed by their harnesses [28], which can substantially improve coding ability relative to simpler harnesses [19].

Context files As coding agents were more broadly adopted, a need arose to provide them with additional context about the codebases they were working with [7, 31]. To this end, model and agent developers recommend including *context files*, such as `AGENTS.md` or `CLAUDE.md`, with codebases [4, 24]. Many agent harnesses even provide built-in commands to initialize such context files automatically using the coding agent itself, e.g., by providing a dedicated `/init` command in the agent interface [3, 26, 28]. At the time of writing, AGENTS.md [1] report that over 60'000 public GitHub repositories include a context file.

Evaluating context files Prior work collected and categorized the content of context files [9, 20], deriving mostly descriptive metrics about their content without investigating their effectiveness [23]. While individual developers report anecdotal evidence of better alignment and solution capabilities when providing context files [30, 31], we are the first to investigate the impact of actively used context files on agent behavior and performance at scale.

Repository-level evaluation Spearheaded by Jimenez et al. [15], evaluating coding agents on the autonomous resolution of real-world repository-level tasks quickly became the gold standard for assessing their capabilities. While initial work focuses on issue resolution [15], follow-up work proposed benchmarks on feature addition [12, 17], unit test generation [21], function generation [18], code performance [14], and security [10]. Our work evaluates whether autonomous issue resolution and feature addition capabilities improve with actively used context files.

Orthogonally, benchmarks have also been extended by mining more recent and more difficult problems [6, 40], as well as instances focusing on end-user applications [36]. We follow their approaches to mining novel task instances to obtain a set of tasks in repositories that feature context files.

3 CTXBENCH

In this section, we discuss the requirements for CTXBENCH, a SWE-BENCH-like benchmark that targets the evaluation of developer-provided context files, its generation process, and its statistics.

3.1 Notation and Definitions

We first introduce the notation to describe codebases, their test suites, and changes to these codebases in the form of patches. Following the notation of Mündler et al. [21], we denote a codebase, or repository, R after applying patch X as $R \circ X$. Several patches can be applied sequentially, i.e., $R \circ X \circ Y$ is the codebase R after applying a first patch X and then a second one Y.

A *test suite* $\mathcal{T}$ is a collection of tests that is used to validate the functionality of code in the repository. Executing a test suite $\mathcal{T}$ on repository state R returns $\text{exec}_R(\mathcal{T}) \in \{\text{PASS}, \text{FAIL}\}$, indicating either that all tests in the suite passed or that at least one test failed. An *issue* I is a task for autonomous completion by the coding agent, such as resolving a bug or implementing a requested feature. We denote quadruples of $(I, R, \mathcal{T}, X^*)$ as *instances*, where the coding agent is tasked with predicting a patch $\hat{X}$ given issue I and repository state R such that $\text{exec}_{R \circ \hat{X}}(\mathcal{T}) = \text{PASS}$, and X^* is the golden patch for that instance. We define the *success rate* $\mathcal{S}$ as the percentage of predicted patches $\hat{X}_i$ for instances $(I_i, R_i, \mathcal{T}_i, X_i^*)$ where $\text{exec}_{R_i \circ \hat{X}_i}(\mathcal{T}_i) = \text{PASS}$.

3.2 Generation of CTXBENCH Instances

Пятиэтапный процесс создания бенча

To construct CTXBENCH, we use a five-stage construction process summarized below. We defer all the prompts used for this process to §E.

Requirements We aim to evaluate the impact of both automatically generated context files and developer-written context files on the success rate of coding agents on real-world tasks and code-

bases. The primary source for real-world codebases is open-source projects and their publicly tracked and documented changes, so-called pull requests (PRs). In order to obtain developer-written context files, we need to source PRs from projects that adopted context files. This is challenging, because context files were only formalized in August 2025, and have not been frequently used before. Further, the adoption of context file is not uniform across the industry: even at the time of writing, many repositories do not include context files.

Finding repositories We first use GitHub search to build a list of potential candidate repositories to extract instances from. Specifically, we select codebases that contain a context file such as `AGENTS.md` or `CLAUDE.md` at the root directory. Next, we filter down to those using Python as the main language and featuring a test suite. Finally, we filter for projects with many publicly documented changes, requiring at least 400 PRs. This criterion allows us to select codebases from which we can extract at least 10 instances after our rigorous post-processing.

Filtering pull requests Given a repository, we filter PRs to retain those that are most likely to generate high-quality instances using a combination of rule-based checks and an LLM agent. We only keep PRs that satisfy the following two criteria: they should reference at least one issue, and they should modify at least one Python file. Further, we filter for PRs that are assessed by the agent to introduce deterministic, testable behaviors that are suitable for SWE-BENCH-like regression tests. We notice that, because the use of context files is a recently emerging trend, most repositories containing context files are niche. These niche repositories have less strict rules regarding pull requests, and thus most PRs may not include specific tests. To enable building instances from these more niche repositories, we therefore do not require PRs to edit unit tests that validate the code changes but instead generate new ones (see below), in contrast to SWE-BENCH, which focused on large and popular repositories and requires PRs to contain unit tests.

Environment Set-Up For every PR and corresponding repository state, we set up an execution environment such that its test suite can be run, using a coding agent. Specifically, we ask the agent to produce a small script that i) sets up the execution environment, ii) runs the test suite and iii) stores the results as a machine-readable dictionary at the root of the repository. We only keep PRs where the resulting dictionary contains at least one passing test, which corresponds to 87% of the filtered instances.

Task Descriptions Many of the smaller repositories we used to source CTXBENCH do not enforce strict requirements on the quality of PR and issue descriptions. As a result, many issues are too imprecise and underspecified to solve the task in a testable manner (e.g., in some cases, the PR body is empty). Further, some PRs implement new features, which would require detailed descriptions about expected behavior and interfaces. We therefore use a third LLM agent to produce a standardized and detailed task description I based on the PR description, associated issues if available, and the original patch X^*. This standardized task description is divided into 6 sections: description, steps to reproduce, expected behavior, observed behavior, specification, and additional information. Importantly, we instruct the agent not to leak the solution in the generated task description, and to provide precise specifications. We inspected all generated instances, and found them to have high quality; none of them leaked the solution, whereas they showed sufficient specificity to correctly solve the given task.

Generating Unit Tests As most collected PRs do not modify or add unit tests that we could use to check the correctness of a given implementation, we use an LLM agent to generate such unit tests. We provide the agent with the standardized task description I, the test files modified by the PR, if available, the original code changes X^* made by the PR, and the base state of the repository R. We then ask it to generate tests that pass for any implementation that resolves the described task. We verify that the added tests fail on R and pass on $R \circ X^*$. Finally, we manually review the generated tests and improve those overfitted to the ref-

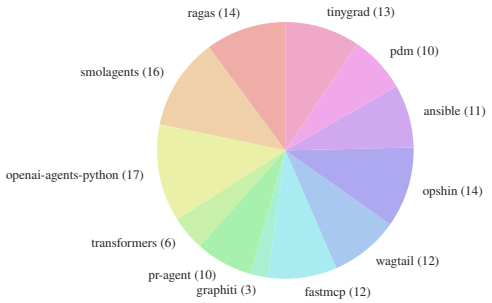


Figure 2: Distribution of CTXBENCH instances across 12 open-source repositories.

erence implementation, resulting in newly generated tests $\mathcal{T}_i^X$. We further determine all tests of the repository test suite $\mathcal{T}_i^R$ that pass on the patched code, i.e., the maximal set $\mathcal{T}_i^{R*} \subseteq \mathcal{T}_i^R$, such that $\text{exec}_{R_i \circ X_i^*}(\mathcal{T}_i^{R*}) = \text{PASS}$, and obtain the final test set $\mathcal{T}_i = \mathcal{T}_i^X \uplus \mathcal{T}_i^{R*}$. These tests achieve an average coverage of 75% of the modified code (Table 1).

Evaluation We thus obtain CTXBENCH instances i, each consisting of a task description I_i, a codebase R_i, golden patch X_i^*, and a set of tests $\mathcal{T}_i$. During evaluation, we first set up the environment before prompting the coding agent with the task description I_i, retrieving the predicted patch $\hat{X}_i$, and measuring $\text{exec}_{R_i \circ \hat{X}_i}(\mathcal{T}_i)$.

Table 1: Average, minimum, and maximum of key statistics of CTXBENCH across the 138 instances. For context files, a section is the content between Markdown headers.

		Mean	Min	Max
PR body	# words	415.3	5	4961
Issue I	# words	211.6	96	500
Codebase	# files	3337	151	26602
PR patch	# lines edited	118.9	12	1973
	# files edited	2.5	1	23
Test	Coverage	75%	2.5%	100%
Context file	# words	641.0	24	2003
	# sections	9.7	1	29

Overview of CTXBENCH Using this process, we obtained 138 instances from a total of 5694 PRs from 12 repositories that meet our criteria, using GPT-5.2 with CODEX as the agent. We visualize the distribution over repositories in Figure 2 and show key statistics of CTXBENCH in Table 1. In comparison to SWE-BENCH, our dataset is both more evenly distributed over repositories and has otherwise similar statistics. We validate the quality of the CTXBENCH instances in §C. Concretely, we find that the developer-provided context files from CTXBENCH are significantly different from LLM-generated ones and thus likely to be human-written or edited. Additionally, we apply the LLM-aided task refinement to SWE-BENCH, showing that this process results in well-specified descriptions and tests, resulting in a fairer benchmark than the unrefined SWE-BENCH.

4 Experimental Evaluation

In this section, we investigate the effect of both LLM-generated and developer-provided context files on coding agent performance across models and agent harnesses.

4.1 Experimental Setup

We describe the experimental setup below, deferring further details to §A.1.

Coding Agents We consider four coding agents, paired with suitable models: CLAUDE CODE [3] with SONNET-4.5 [5], CODEX [26] with GPT-5.2 and GPT-5.1 MINI [32], and QWEN CODE [28] with QWEN3-30B-CODER [34]. For CLAUDE CODE, we use the default settings and set the temperature of SONNET-4.5 to 0. Similarly, for CODEX, we also use the default settings and set the temperature of GPT-5.2 and GPT-5.1 MINI to 0. For QWEN CODE, we enable chat compression upon reaching 60% of the total context limit (set to 256K tokens), restrict shell outputs to 2000 tokens, and set the temperature of QWEN3-30B-CODER to 0.7 with top-p sampling at 0.8. We deploy QWEN3-30B-CODER locally using vLLM [16]. We sample completions for each agent once. For all agents, the context file is fed into their context, either by writing it to `AGENTS.md` for CODEX and QWEN CODE, or to `CLAUDE.md` for CLAUDE CODE.

Datasets We use the LITE split of SWE-BENCH [15], which consists of 300 tasks sourced from GitHub issues across 11 popular Python repositories, none containing developer-provided context files, and our novel CTXBENCH, consisting of 138 instances from 12 repositories, all containing developer-provided context files (see §3).

Settings We consider three context file settings:

- NONE No context file is available, i.e., we remove developer-provided files for CTXBENCH.
- LLM An LLM-generated context file is available. We use the recommended initialization command and model for each agent individually to generate the context file using the pre-patch repository state R.
- DEV A developer-provided context file is available. We use the context file of the pre-patch repository state R. Only available for CTXBENCH.

Очень интересные результаты

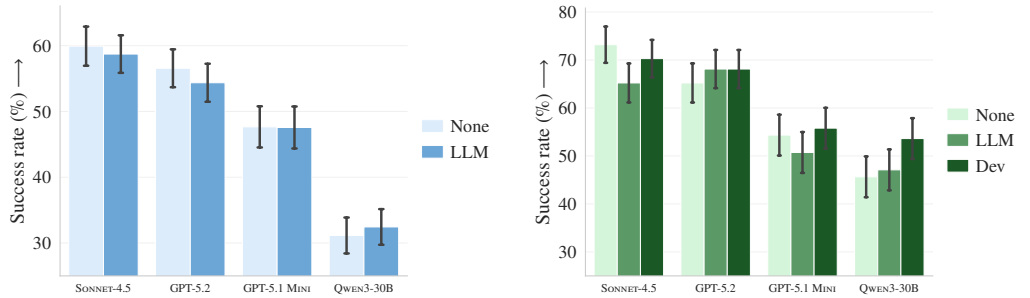


Figure 3: Resolution rate for 4 different models, without context files, with LLM-generated context files, and with developer-written context files, on SWE-BENCH (left) and CTXBENCH (right).

Table 2: The average number of steps (lower is better) and execution cost (in USD, lower is better) per SWE-Bench Lite and CTXBENCH instance without context files (None), with LLM-generated context files (LLM), and with developer-context files (DEV.). We **bold** the best setting.

	Type	SONNET-4.5		GPT-5.2		GPT-5.1 M.		QWEN3-30B	
		Steps	Cost	Steps	Cost	Steps	Cost	Steps	Cost
SWE-BENCH	None	**54.4 ± 2.2**	**1.30 ± 0.07**	**12.5 ± 0.8**	**0.32 ± 0.05**	**40.9 ± 4.2**	**0.18 ± 0.03**	**29.7 ± 1.9**	**0.12 ± 0.01**
	LLM	57.2 ± 2.3	1.51 ± 0.08	12.7 ± 0.8	0.43 ± 0.05	45.2 ± 4.2	0.22 ± 0.03	32.2 ± 1.9	0.13 ± 0.01
CTX-BENCH	None	**40.7 ± 3.0**	**1.15 ± 0.10**	**12.1 ± 1.6**	**0.38 ± 0.11**	**40.6 ± 6.7**	**0.18 ± 0.05**	**31.5 ± 3.1**	**0.13 ± 0.02**
	LLM	46.5 ± 3.9	1.33 ± 0.14	13.1 ± 1.5	0.57 ± 0.18	46.9 ± 6.4	0.20 ± 0.04	34.2 ± 3.4	0.15 ± 0.02
	DEV.	45.3 ± 3.6	1.30 ± 0.13	13.6 ± 1.7	0.54 ± 0.23	46.6 ± 7.1	0.19 ± 0.04	32.8 ± 3.7	0.15 ± 0.03

Metrics The main metric for agent performance is success rate (§3.1), i.e., the portion of instances for which the agent produces a patch that leads to all tests passing. We additionally consider the number of *steps* the agent requires to complete a task. Each step is one interaction with the environment, e.g., calling a shell tool or modifying a file. Finally, we report the total *cost* of LLM inference required to complete a task. For QWEN3-30B-CODER, we estimate the cost from the average OpenRouter API price. We report the statistical significance of changes in these metrics in Tables 3 and 6.

4.2 Main Results: Context Files Don't Improve Performance but Increase Cost

LLM-generated context files increase cost and don't improve performance LLM-generated context files cause performance drops in 5 out of 8 settings across SWE-BENCH and CTXBENCH (see Figure 3). Specifically, the average resolution rate is reduced by 0.5% and 2% on average on SWE-BENCH and CTXBENCH, respectively. With p-values of 87% and 37% (see Table 3) under a two-sided test, this indicates that they have no significant effect on performance. Meanwhile, they increase the # steps in every setting, on average by 2.45 and 3.92, respectively, leading to a significant (p-value < 0.001%) cost increase of 20% and 23% on average, respectively (see Table 2).

Table 3: Two-sided Cochran-Mantel-Haenszel test comparing the effect of different context files on the resolution rate. The null hypothesis is that resolution rates are equal. We **bold** significant p-values.

Benchmark	Comparison	p-value
SWE-BENCH	None vs LLM	0.87
CTXBENCH	None vs LLM	0.37
CTXBENCH	None vs Dev	0.21
CTXBENCH	**LLM vs Dev**	**0.038**

Developer-provided context files increase cost and outperform LLM-generated ones Developer-provided context files improve agent performance by 2.4% on average ($p = 21\%$), significantly outperforming LLM-generated ones ($p = 3.8\%$) (see Table 3). Despite not being agent-specific, they improve performance for all agents but CLAUDE CODE (see Figure 3 right). However, developer-provided context files also increase the average number of steps and the cost, on average by 3.34 steps and at most 19%, respectively.

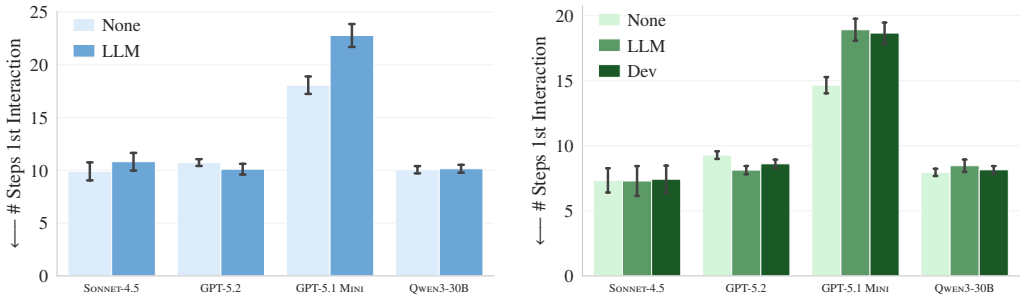


Figure 4: Number of steps before the first interaction between the agent and a file included in the PR patch (lower is better) is generally lower without context files than with LLM-generated context files or with developer-written context files (Human) on SWE-BENCH (left) and CTXBENCH (right).

4.3 Understanding Behavioral Changes Induced by Context Files

In this section, we investigate why context files do not affect performance meaningfully while increasing costs. We find that, while instructions provided in context files are well followed, explaining the increase in cost, they do not provide effective repository overviews.

Context files do not provide effective overviews One recommendation for context files is to include a codebase overview [1]. Across the 12 developer-provided context files in CTXBENCH, 8 include a dedicated codebase overview, with 4 explicitly enumerating and describing the directories and subdirectories in the repository. Similarly, both the CODEX and QWEN CODE context file generation prompts explicitly instruct the agent to include an overview section, while the CLAUDE CODE prompt advocates for a high-level overview only and warns against listing components that are easily discoverable. We use GPT-OSS-120B as a judge to assess which of the LLM-generated context files contain codebase overviews. Surprisingly, 100% of SONNET-4.5-generated context files are flagged for overviews, and 95% and 99% are flagged for QWEN3-30B-CODER and GPT-5.2, respectively. Only GPT-5.1 MINI has significantly fewer overviews (36%).

To assess the usefulness of these overviews, we measure how quickly agents discover files relevant to the described issue I. In particular, we measure the average number of steps before the coding agent interacts with any file modified in the original PR patch X^*. We exclude the 3% of instances in which the agent never interacts with any file modified in X^*. Both on SWE-BENCH and CTXBENCH, and for all agents, the context files do not meaningfully reduce this metric, while increasing the number of required steps significantly for GPT-5.1 MINI, as shown in Figure 4.

Inspecting the traces of GPT-5.1 MINI manually, we observe that the significant increase in the required number of steps is due to it (i) issuing multiple commands to find the context files and (ii) reading them (multiple times) despite them being already included in the agent's context. Interestingly, we only observed this behavior if context files were present at all. We conclude that context files are not effective at providing a repository overview.

Context files lead to more testing, exploration, and specialized tool use Many context files contain recommendations on how to interact with the repository when solving issues. To inspect whether agents follow these, we analyze the type and frequency of tool calls in agent traces. For tools included in the agentic framework (e.g., Read, Write, or Todo), we simply record the name of the tool being called. For shell commands, we use an LLM-as-a-judge approach (using GPT-OSS-120B) to extract the executed commands (e.g., `uv`, `pytest`, `cat`) and to categorize the intent of the tool call (e.g., install dependencies, run tests, read files). We build these categories iteratively, starting with an empty set, and allowing the judge to create new categories as needed (see §A).

In Figure 5, we show the increase in average tool use when including LLM-generated (bright green) or developer-provided (dark green) context files. Negative values imply a decrease in tool use. We find that, across all models, when context files are present, the coding agents run more tests. They also tend to navigate the repository more: they search more files (`grep`), read more files, and write more files. Lastly, adding context files causes agents to use more repository-specific tooling (e.g., `uv`

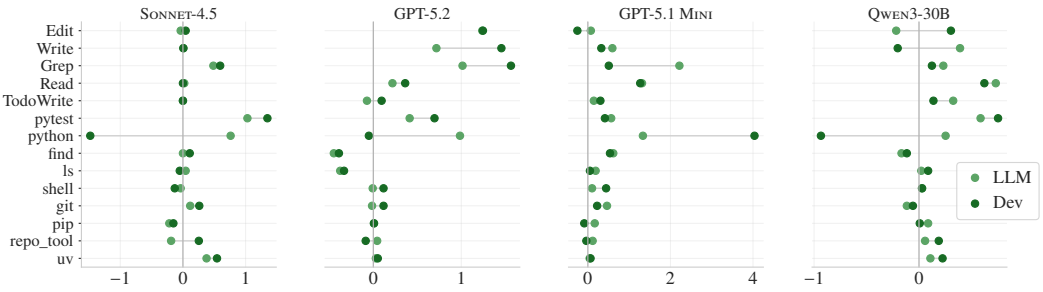


Figure 5: Increase in average tool use with LLM-generated (bright green) or developer-provided (dark green) context files, compared to the average tool use without context files. We map CODEX and QWEN CODE tools to the CLAUDE CODE equivalents (we detail the mapping in §A).

and `repo_tool`). In Figure 11 (§A), we perform a similar analysis using the intent of the tool call, and our conclusion is the same: context files lead to more testing and exploration.

With respect to recommendations to use specific tools, we find that agents generally abide by them. For instance, `uv` is used 1.6 times per instance on average when mentioned in the context files, compared to fewer than 0.01 times when it is not mentioned, and repository-specific tools are used 2.5 times per instance on average when mentioned, compared to fewer than 0.05 times when they are not mentioned. In particular, this result implies that the absence of improvements when using context files is not due to a lack of instruction-following capabilities. We defer a more in-depth analysis to §E.2.

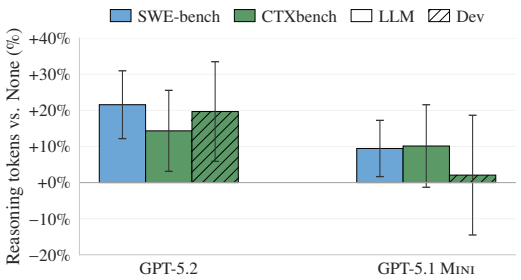


Figure 6: Relative change in average reasoning tokens spent by GPT-5.2 and GPT-5.1 MINI compared to the no-context files baseline (None), computed on matched instances.

Following context files requires more thinking We hypothesize that these additional instructions make the task harder. We analyze the average number of reasoning tokens used by GPT-5.2 and GPT-5.1 MINI, as their adaptive reasoning [25] allows them to use more reasoning tokens for tasks that they deem harder. Figure 6 shows the paired relative change compared to the no-context files baseline: LLM-generated context files increase the average number of reasoning tokens by 22% for GPT-5.2 and 10% for GPT-5.1 MINI on SWE-BENCH (respectively 14% and 10% on CTXBENCH), and developer-provided context files increase it by 20% for GPT-5.2 and 2% for GPT-5.1 MINI.

4.4 Ablations

To investigate whether the poor performance of LLM-generated context files stems from a lack of model capabilities or bad prompts, we analyze differences between the context files generated by different models and the impact of the prompt used to create the context files.

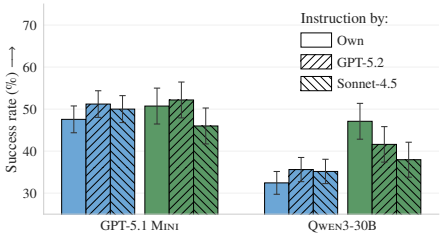


Figure 7: Comparison of context files effect across different generating models.

Stronger models don't generate better context files We compare context files generated with GPT-5.2 + CODEX and SONNET-4.5 + CLAUDE CODE against self-generated ones when used by GPT-5.1 MINI and QWEN3-30B-CODER. While performance improves on SWE-BENCH (2% on average), it worsens on CTXBENCH (3% on average), showing that stronger models do not necessarily generate superior context files.

Противоречит стандартному подходу с инструкцией AGENTS.md самой мощной модели

No difference between the specific prompts We compare context files generated using the prompts of CODEX and CLAUDE CODE across all agents and models in Figure 8. Surprisingly, CLAUDE CODE performs better with context files generated using the CODEX prompt, while both GPT-5.2 and GPT-5.1 MINI perform better on SWE-BENCH with the CODEX prompt but worse on CTXBENCH. Overall, neither the prompt that matches the underlying agent nor a specific prompt performs consistently best, indicating that sensitivity to good prompts is overall small.

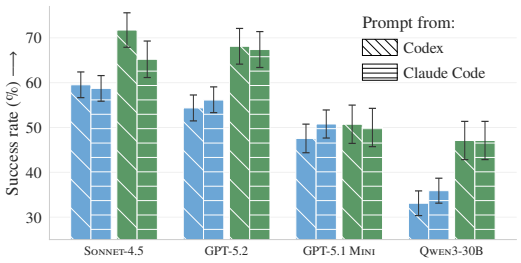


Figure 8: Ablation study on which prompt is used to generate context files.

Neither context file length nor specific instruction categories have a strong effect on performance In §B, we show that the length of context files has no significant impact on our results, that removing specific categories from LLM-generated context files does not change the results, and that contamination with task-specific knowledge cannot explain why context files do not improve performance.

5 Limitations and Future Work

Programming languages The current evaluation is focused on Python. Since this is a language that is widely represented in the training data, detailed knowledge about tooling, dependencies, and other repository specifics might be present in the models' parametric knowledge, nullifying the effect of context files. Future work may investigate the effect of context files on more niche programming languages and toolchains that are less represented in the training data [8, 27].

Context files beyond task resolution In this work, we evaluate the impact of context files on task resolution rate. However, other aspects of coding agent performance, such as code efficiency [14] and security [10], would be interesting directions for future work. Security, specifically, was found to be sensitive to prompt changes in prior work [37].

Improving context file generation Another interesting avenue opened by this work is how to improve the automatic generation of *useful* context files. Several related works in the direction of planning and continuous learning from prior tasks may be applicable to this task [11, 33, 41]. By tackling this challenge, future agents could gain a long-term capability at meaningful self-improvement. Our work may serve as a baseline for how to rigorously evaluate automatically generated context files.

6 Conclusion

We evaluate the impact of context files on coding agent performance for four common coding agents on SWE-BENCH and the novel CTXBENCH, built from recent GitHub issues and less popular repositories containing developer-written context files. We find that all context files consistently increase the cost and number of steps required to complete tasks. LLM-generated context files have a marginal negative effect on task success rates, while developer-written ones provide a marginal performance gain, neither statistically significant.

Our trace analyses show that instructions in context files are generally followed and lead to more testing and broader exploration; however, they do not function as effective repository overviews. Overall, our results suggest that context files don't improve coding agent performance, and should only contain specific additional instructions beyond what is already available in the codebase. This highlights a concrete gap between current agent-developer recommendations and observed outcomes, and motivates future work on principled ways to automatically generate concise, task-relevant guidance for coding agents.

References

[1] AGENTS.md. AGENTS.md - A simple, open format for guiding coding agents, 2025. URL https://agents.md/.

[2] Aider. AI pair programming in your terminal, 2024. URL https://aider.chat.

[3] Anthropic. Claude Code overview, 2025. URL https://code.claude.com/docs/en/overview.

[4] Anthropic. Using CLAUDE.md files: Customizing Claude Code for your codebase, 2025. URL https://claude.com/blog/using-claude-md-files.

[5] Anthropic. Claude Sonnet 4.5, 2025. URL https://www.anthropic.com/news/claude-sonnet-4-5.

[6] Ibragim Badertdinov, Alexander Golubev, Maksim Nekrashevich, Anton Shevtsov, Simon Karasik, Andrei Andriushchenko, Maria Trofimova, Daria Litvintseva, and Boris Yangel. SWE-rebench: An Automated Pipeline for Task Collection and Decontaminated Evaluation of Software Engineering Agents. *arXiv preprint*, 2025. URL https://arxiv.org/abs/2505.20411.

[7] Gowtham Boyina. Why I Created AGENTS.md: A Simple Solution to a Growing Problem, 2025. URL https://thegowtham.medium.com/why-i-created-agents-md-a-simple-solution-to-a-growing-problem-3afc1f6211f7.

[8] Federico Cassano, John Gouwar, Daniel Nguyen, Sydney Nguyen, Luna Phipps-Costin, Donald Pinckney, Ming-Ho Yee, Yangtian Zi, Carolyn Jane Anderson, Molly Q Feldman, et al. MultiPL-E: A Scalable and Extensible Approach to Benchmarking Neural Code Generation. *arXiv preprint*, 2022. URL https://arxiv.org/abs/2208.08227.

[9] Worawalan Chatlatanagulchai, Hao Li, Yutaro Kashiwa, Brittany Reid, Kundjanasith Thonglek, Pattara Leelaprute, Arnon Rungsawang, Bundit Manaskasemsak, Bram Adams, Ahmed E. Hassan, et al. Agent READMEs: An Empirical Study of Context Files for Agentic Coding. *arXiv preprint*, 2025. URL https://arxiv.org/abs/2511.12884.

[10] Junkai Chen, Huihui Huang, Yunbo Lyu, Junwen An, Jieke Shi, Chengran Yang, Ting Zhang, Haoye Tian, Yikun Li, Zhenhao Li, et al. SecureAgentBench: Benchmarking Secure Code Generation under Realistic Vulnerability Scenarios. *arXiv preprint*, 2025. URL https://arxiv.org/abs/2509.22097.

[11] Yang Cheng, Zilai Wang, Weiyu Ma, Wenhui Zhu, Yue Deng, and Jian Zhao. EvoCurr: Self-evolving Curriculum with Behavior Code Generation for Complex Decision-making. *arXiv preprint*, 2025. URL https://arxiv.org/abs/2508.09586.

[12] Yaxin Du, Yuzhu Cai, Yifan Zhou, Cheng Wang, Yu Qian, Xianghe Pang, Qian Liu, Yue Hu, and Siheng Chen. SWE-Dev: Evaluating and Training Autonomous Feature-Driven Software Development. *arXiv preprint*, 2025. URL https://arxiv.org/abs/2505.16975.

[13] Google. google-gemini/gemini-cli: An open-source AI agent that brings the power of Gemini directly into your terminal, 2025. URL https://github.com/google-gemini/gemini-cli.

[14] Xinyi He, Qian Liu, Mingzhe Du, Lin Yan, Zhijie Fan, Yiming Huang, Zejian Yuan, and Zejun Ma. SWE-Perf: Can Language Models Optimize Code Performance on Real-World Repositories? *arXiv preprint*, 2025. URL https://arxiv.org/abs/2507.12415.

[15] Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R. Narasimhan. SWE-bench: Can Language Models Resolve Real-world Github Issues? In *ICLR*, 2024. URL https://openreview.net/forum?id=VTF8yNQM66.

[16] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient Memory Management for Large Language Model Serving with PagedAttention. In *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*, 2023.

[17] Wei Li, Xin Zhang, Zhongxin Guo, Shaoguang Mao, Wen Luo, Guangyue Peng, Yangyu Huang, Houfeng Wang, and Scarlett Li. FEA-Bench: A Benchmark for Evaluating Repository-Level Code Generation for Feature Implementation. In *ACL*, 2025. URL https://aclanthology.org/2025.acl-long.839/.

[18] Shanchao Liang, Yiran Hu, Nan Jiang, and Lin Tan. Can Language Models Replace Programmers for Coding? REPOCOD Says 'Not Yet'. *arXiv preprint*, 2024. URL https://arxiv.org/abs/2410.21647.

[19] Kilian Lieret, Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. SWE-bench Bash Only, 2025. URL https://www.swebench.com/bash-only.html.

[20] Seyedmoein Mohsenimofidi, Matthias Galster, Christoph Treude, and Sebastian Baltes. Context Engineering for AI Agents in Open-Source Software. *arXiv preprint*, 2025. URL https://arxiv.org/abs/2510.21413.

[21] Niels Mündler, Mark Niklas Müller, Jingxuan He, and Martin T. Vechev. SWT-Bench: Testing and Validating Real-World Bug-Fixes with Code Agents. In *NeurIPS*, 2024. URL http://papers.nips.cc/paper_files/paper/2024/hash/94f093b41fc2666376fb1f667fe282f3-Abstract-Conference.html.

[22] Christian Mürtz and Mark Niklas Müller. Agents in the wild - dashboard. https://insights.logicstar.ai, 2025. URL https://doi.org/10.5281/zenodo.15846865. Interactive web dashboard. Code available at https://github.com/logic-star-ai/insights.

[23] Matt Nigh. How to write a great agents.md: Lessons from over 2,500 repositories, 2025. URL https://github.blog/ai-and-ml/github-copilot/how-to-write-a-great-agents-md-lessons-from-over-2500-repositories/.

[24] OpenAI. OpenAI co-founds the Agentic AI Foundation under the Linux Foundation, 2025. URL https://openai.com/index/agentic-ai-foundation/.

[25] OpenAI. GPT-5.1: A smarter, more conversational ChatGPT, 2025. URL https://openai.com/index/gpt-5-1/.

[26] OpenAI. openai/codex: Lightweight coding agent that runs in your terminal, 2025. URL https://github.com/openai/codex.

[27] Gabriel Orlanski, Kefan Xiao, Xavier Garcia, Jeffrey Hui, Joshua Howland, Jonathan Malmaud, Jacob Austin, Rishabh Singh, and Michele Catasta. Measuring the Impact of Programming Language Distribution. In *ICML*, 2023. URL https://proceedings.mlr.press/v202/orlanski23a.html.

[28] QwenLM. QwenLM/Qwen3-Coder: Qwen3-Coder is the code version of Qwen3, the large language model series developed by Qwen team, Alibaba Cloud, 2025. URL https://github.com/QwenLM/Qwen3-Coder.

[29] Suproteem K Sarkar. Ai agents, productivity, and higher-order thinking: Early evidence from software development. *Available at SSRN 5713646*, 2025.

[30] Paul Sawers. The Rise of Agents.md, an Open Standard and Single Source of Truth for AI Coding Agents, 2025. URL https://tessl.io/blog/the-rise-of-agents-md-an-open-standard-and-single-source-of-truth-for-ai-coding-agents/.

[31] Steve Sewell. Improve your AI code output with AGENTS.md (+ my best tips), 2025. URL https://www.builder.io/blog/agents-md.

[32] 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*, 2026. URL https://arxiv.org/abs/2601.03267.

[33] Mirac Suzgun, Mert Yuksekgonul, Federico Bianchi, Dan Jurafsky, and James Zou. Dynamic Cheatsheet: Test-Time Learning with Adaptive Memory. *arXiv preprint*, 2025. URL https://arxiv.org/abs/2504.07952.

[34] Qwen Team. Qwen3 Technical Report. *arXiv preprint*, 2025. URL https://arxiv.org/abs/2505.09388.

[35] Katherine Thai, Bradley Emi, Elyas Masrour, and Mohit Iyyer. Editlens: Quantifying the extent of ai editing in text, 2025. URL https://arxiv.org/abs/2510.03154.

[36] Konstantinos Vergopoulos, Mark Niklas Müller, and Martin Vechev. Automated Benchmark Generation for Repository-Level Coding Tasks. *arXiv preprint*, 2025. URL https://arxiv.org/abs/2503.07701.

[37] Mark Vero, Niels Mündler, Victor Chibotaru, Veselin Raychev, Maximilian Baader, Nikola Jovanovic, Jingxuan He, and Martin T. Vechev. BaxBench: Can LLMs Generate Correct and Secure Backends? In *ICML*, 2025.

[38] Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, et al. OpenHands: An Open Platform for AI Software Developers as Generalist Agents. In *ICLR*, 2025. URL https://openreview.net/forum?id=OJd3ayDDoF.

[39] John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering. In *NeurIPS*, 2024. URL http://papers.nips.cc/paper_files/paper/2024/hash/5a7c947568c1b1328ccc5230172e1e7c-Abstract-Conference.html.

[40] Linghao Zhang, Shilin He, Chaoyun Zhang, Yu Kang, Bowen Li, Chengxing Xie, Junhao Wang, Maoquan Wang, Yufan Huang, Shengyu Fu, et al. SWE-bench Goes Live! *arXiv preprint*, 2025. URL https://arxiv.org/abs/2505.23419.

[41] Qizheng Zhang, Changran Hu, Shubhangi Upasani, Boyuan Ma, Fenglu Hong, Vamsidhar Kamanuru, Jay Rainton, Chen Wu, Mengmeng Ji, Hanchen Li, et al. Agentic Context Engineering: Evolving Contexts for Self-Improving Language Models. *arXiv preprint*, 2025. URL https://arxiv.org/abs/2510.04618.

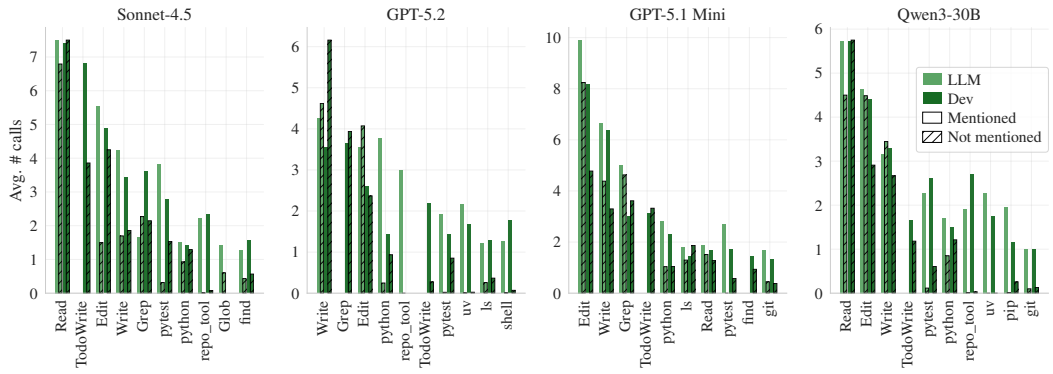


Figure 9: Average number of tool calls depending on whether the tool name is mentioned in the context files. For tool names, we use the equivalence classes from Table 4, and consider a tool to be mentioned in the context file if any tool from the corresponding equivalence class is mentioned in the context file.

A Experimental Details

In this section, we provide additional details about our experiments from §4.

A.1 Additional Experimental Details

We now describe the remaining experimental details for the experiments in §4.2.

Coding environment For CTXBENCH instances, we run the coding agent in a Docker container with basic tooling (python, apt-get, uv, ...) and Internet access. Importantly, we remove the git commit history and all remotes. For SWE-BENCH, we use the Docker images provided by Jimenez et al. [15]. We let the coding agents access the web (either via dedicated tools or through the command line) and manually checked that the agents do not cheat (e.g., looking at the PR corresponding to the instance description). We find no such case of cheating, and web access represents a minority of the tool calls (less than 1%). For CLAUDE CODE, we keep the Task tool enabled: it allows SONNET-4.5 to invoke sub-agents, using HAIKU-4.5, to solve sub-tasks. For instance, a sub-task can involve exploring the repository to find specific files.

A.2 Trace Analysis

Here, we detail the experiments from §4.3. In particular, we give the mapping used to aggregate tool names across coding agents, analyze the correlation between the number of tool uses and whether the tool is mentioned in the context files, and expand the trace analysis to the intent behind tool calls.

Experimental setup We recall the experimental setup from §4.3. Given a list of tool calls from an agent, we analyze the frequency of each tool call. For tools included in the agentic framework (e.g., Read, Write, or Todo), we simply record the name of the tool being called. For shell commands, we use an LLM-as-a-judge to extract (from the command and its output) the concrete command that was executed (e.g., `uv`, `pytest`, `cat`) and to categorize the intent of the tool call (e.g., install dependencies, run tests, read files). We build the categories iteratively. We start with an empty set of categories, and for each shell command, we ask the judge to assign it to an existing category if possible and otherwise create a new category. As the judge, we use GPT-OSS-120B, and the prompt is given in §E.2. Finally, we manually merge duplicate and closely related categories.

Table 4: Equivalence classes used to group the different tool calls.

CLAUDE CODE tool	CODEX	QWEN CODE
Edit	`sed`	`sed`, `edit`
Write	`apply_patch`	`write_file`
Grep	`grep`, `rg`	`grep`
Read	`cat`	`cat`, `read_file`, `search_file_content`
TodoWrite	`update_plan`	`todo_write`

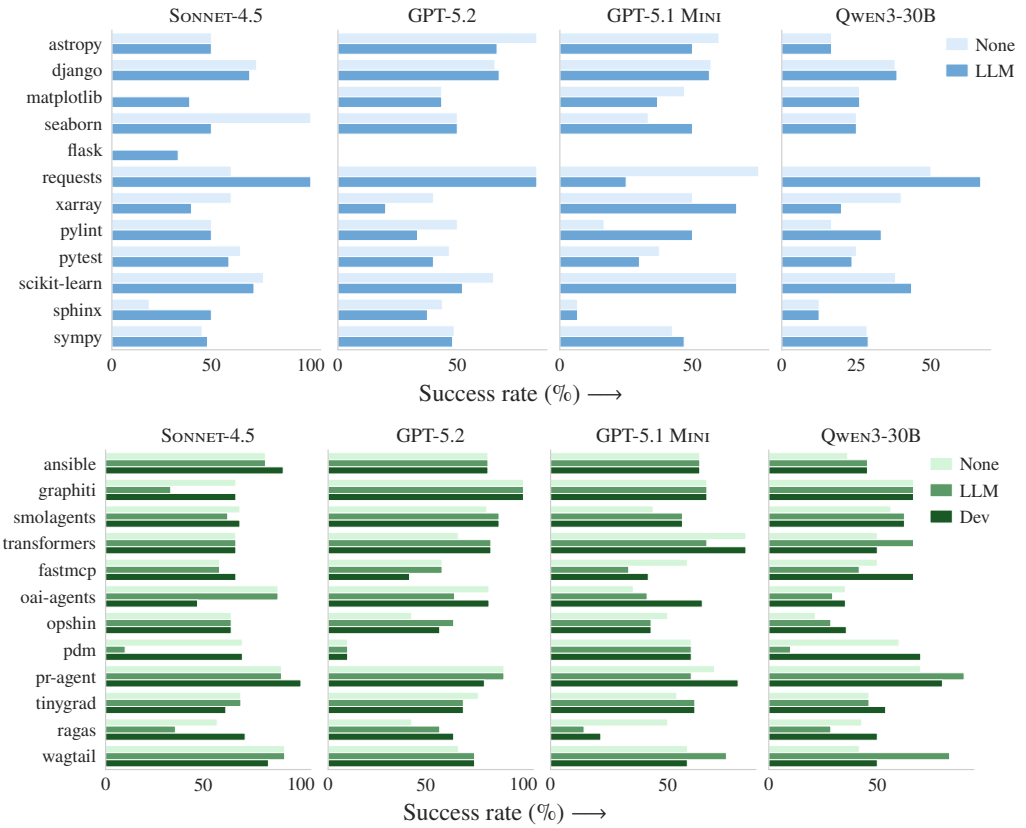


Figure 10: Resolution rate grouped by repository for four different models: without context files, with LLM-generated context files, and with developer-written context files on SWE-bench (top) and CTXbench (bottom). For SWE-bench in particular, the majority of instances come from the same repository (`django`), making per-repository estimates of the success rate noisy.

Refining the tool names For Figure 5, we further manually refined the tool names for readability. In particular, in Table 4, we map tool names from other agents, namely Codex and Qwen Code, as well as some CLI tools, to Claude Code tooling. Lastly, for repository-specific tooling (e.g., `pdm`, `ansible`, or `opshin`), we grouped them into the `repo_tool` category.

Correlation between the number of tool calls and context files In Figure 9, we show the average number of tool calls depending on whether the tool name is mentioned in the context file. We find that, if a tool name is mentioned in the context files, this increases its usage by the coding agents. For instance, `uv`, `pytest`, or repository-specific tools (`repo_tool`) are used almost exclusively if they are mentioned in the context file. This means that instructions in the context files are followed, and that a lack of instruction-following capabilities does not explain why we observe, in §4.2, no gain in accuracy when using context files.

Analyzing intents of tool calls For the tool intents extracted with the LLM judge (334 different categories in total), we further aggregate them into the following 10 categories:

- **git**: Repository and version-control operations (e.g., commits, branches, diffs, checkout, stash, status).
- **model**: Model lifecycle tasks such as downloading or loading models, inspecting configurations or parameters, and running inference.
- **env_deps**: Python environment and dependency management (virtualenv or venv, installations, versions, lockfiles).

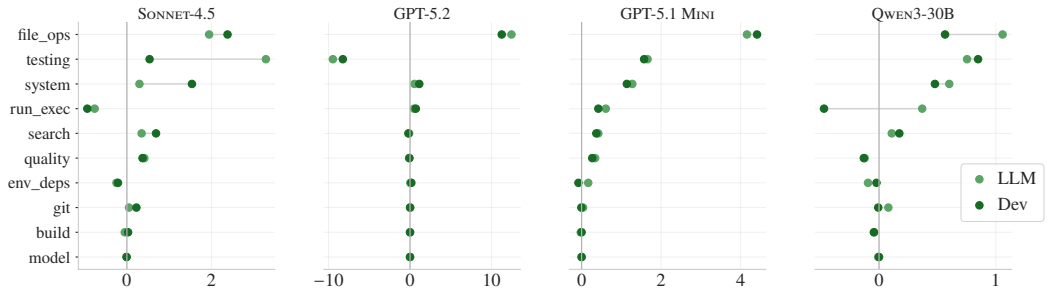


Figure 11: Increase in the average tool use (grouped into high-level categories) when including LLM-generated (bright green) or developer-provided (dark green) context files, compared to the average tool use without context files. For the high-level categories, we use an LLM to categorize the various tool calls.

Table 5: The success rate (in %) plus or minus the standard error per SWE-bench and CTXbench. We **bold** the best setting.

	Type	Sonnet-4.5	GPT-5.2	GPT-5.1 M.	Qwen3-30B
SWE-bench	None	$\mathbf{59.9 \pm 3.0}$	$\mathbf{56.6 \pm 2.9}$	$\mathbf{47.7 \pm 3.1}$	31.1 ± 2.7
	LLM	58.7 ± 2.9	54.4 ± 2.9	47.6 ± 3.2	$\mathbf{32.4 \pm 2.7}$
Plan-bench	None	$\mathbf{73.2 \pm 3.8}$	65.2 ± 4.1	54.3 ± 4.3	45.7 ± 4.3
	LLM	65.2 ± 4.1	$\mathbf{68.1 \pm 4.0}$	50.7 ± 4.3	47.1 ± 4.3
	Dev.	70.3 ± 3.9	$\mathbf{68.1 \pm 4.0}$	$\mathbf{55.8 \pm 4.2}$	$\mathbf{53.6 \pm 4.3}$

- **build**: Building, compiling, or packaging code and producing artifacts or distribution packages.
- **quality**: Code quality and correctness checks (linting, formatting, type checking, validation or verification, schema checks).
- **testing**: Running and reviewing tests (unit, integration, regression, sanity, pytest) and test results.
- **run_exec**: Executing workflows and scripts or commands (Python, shell, Django), including reproduction and debugging runs.
- **search**: Discovery and inspection actions (search, find, grep, glob, list, view, show, display, inspect, parse).
- **file_ops**: Direct file and filesystem operations (read, write, edit, copy, move, delete, create, permissions, paths).
- **system**: System and miscellaneous utilities (processes, disk usage, environment variables, HTTP checks, checksums, tool or device information, help).

In Figure 11, we show the difference in frequency of these categories with and without context files. The conclusion is similar to that of §4.3: the presence of context files significantly increases the number of tests run by coding agents, as well as the extent of codebase exploration and code quality checks.

A.3 Per-repository Success Rate

In Figure 10, we show the success rate of the different scenarios (None, LLM, and Dev) grouped by repository. For both SWE-bench and CTXbench, there is no single repository where the presence of context files has a significant impact. Nonetheless, for CTXbench in particular, we see that the difficulty across instances is relatively balanced, validating our approach to building the instances.

A.4 Additional Data

In Table 5, we report the success rate depicted in Figure 3.

A.5 Additional Statistical Testing

Table 6: Stratified permutation tests comparing the effect of different context files on cost and number of steps. Under the null hypothesis, we assume that the context files do not influence cost (respectively, the number of steps).

Experiment	Benchmark	Metric	Comparison	p-value	Significant
Experiment 1	SWE-BENCH	Steps	None vs LLM	0.0287	Yes
Experiment 1	SWE-BENCH	Cost	None vs LLM	< 0.00001	Yes
Experiment 1	CTXBENCH	Steps	None vs LLM	0.00004	Yes
Experiment 1	CTXBENCH	Steps	None vs Human	0.00002	Yes
Experiment 1	CTXBENCH	Cost	None vs LLM	0.00064	Yes
Experiment 1	CTXBENCH	Cost	None vs Human	0.0126	Yes

In Table 6, we show the results of stratified permutation tests assessing the statistical significance of the increase in steps and costs induced by context files. We find that context files systematically and significantly increase the average number of steps and the cost required to solve instances.

B Additional Results

In this section, we propose an additional evaluation of the effects of context files on coding agents. In particular, we show that context files can act as effective overviews when no documentation is present. We evaluate how the length of context files influences our results, which categories within context files have the most impact, and whether contamination explains why we do not observe performance improvements when using context files.

Context files are redundant documentation We show in §4.2 that context files do not provide an effective repository overview. Our hypothesis is that LLM-generated context files are mostly redundant with existing documentation, while developer-provided context files add additional information. We confirm this hypothesis by manually removing all documentation (files ending with `.md`, example code, and the contents of the `docs/`) after generating the context file and before evaluating the coding agents, excluding CLAUDE CODE for cost reasons, and show the results in Figure 12. In this setting, where context files are the only source of documentation available, we find that LLM-generated context files not only consistently improve performance by 2.7% on average, but also outperform developer-written ones across settings. This may also explain anecdotal evidence reporting that coding agents perform better after adding context files [31], since many less popular repositories contain little to no documentation.

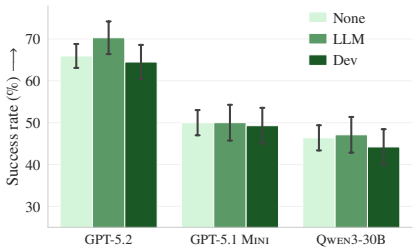


Figure 12: When removing all documentation-related files from the codebase, LLM-generated context files tend to outperform developer-provided (Human) ones on CTXBENCH.

The length of context files does not influence our findings In Figure 13, we bin the success rate and per-instance cost results from Figure 3 and Table 2 by the length of the context file. We observe no clear dependency between the success rate or the per-instance cost and the context file length. This suggests that the length of context files does not influence our findings, and that the increase in cost is better explained by the fact that coding agents tend to follow instructions written in the context files (see §4.3).

No specific categories in LLM-generated context files help significantly We manually inspect LLM-generated context files and identify three categories that appear in most of them: *Overview* (Does the context file explain the codebase structure), *Tooling* (How to set up the environment? Which tools to use?), and *Testing* (How to run tests in the codebase?). For developer-provided context files, there are no clear categories shared between them.

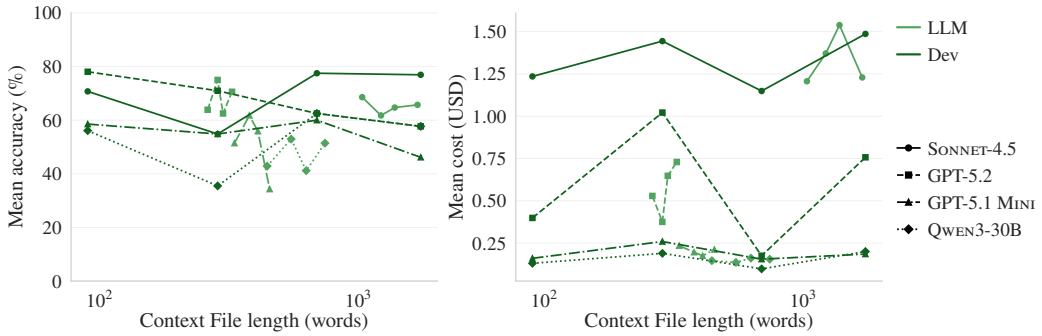


Figure 13: Resolution rate and average cost (in USD) for four different models with LLM-generated context files and with developer-provided ones on CTXBENCH, binned by the length of the context files. We find no correlation between the resolution rate and the length of the context files.

Table 7: Ablating common categories from LLM-generated context files does not significantly improve accuracy on either benchmark. For accuracy, p-values are computed with McNemar's test against the full context file condition; for cost, p-values are computed with a permutation test.

Benchmark	Metric	Full	Without testing	Without overview	Without tooling
CTXBENCH	Accuracy	68.12%	66.67% ($p = 0.80$)	62.32% ($p = 0.15$)	63.77% ($p = 0.31$)
	Cost	\$0.4715	\$0.3730 ($p = 0.023$)	\$0.4027 ($p = 0.24$)	\$0.4815 ($p = 0.97$)
SWE-BENCH	Accuracy	54.36%	57.72% ($p = 0.099$)	54.20% ($p = 0.73$)	53.69% ($p = 0.85$)
	Cost	\$0.3272	\$0.2756 ($p = 0.0035$)	\$0.3018 ($p = 0.10$)	\$0.2715 ($p = 0.0012$)

To assess the importance of each category, we use GPT-5.4 to remove each individual category from the context file, and then rerun SWE-BENCH and CTXBENCH on GPT-5.2. Table 7 shows that the testing category leads to a significant increase in cost in both SWE-BENCH and CTXBENCH, and the tooling category leads to a significant increase in cost for SWE-BENCH. Meanwhile, no category has a significant positive or negative effect on benchmark accuracy. Even if tendencies appear, such that omitting test harness details in SWE-BENCH increases performance, this is contradicted by a slightly negative effect of omitting it on CTXBENCH. Overall, these results back our recommendation that LLM-generated context files do influence behaviors (as seen by the cost increase) but ultimately do not significantly help improve coding agents' accuracy, even when considering only particular parts of the context file.

Contamination does not influence our results To isolate the effect of contamination on our results, we split CTXBENCH instances by whether the PR predates each model's knowledge cutoff and compute accuracy for all three settings: no context files, LLM-generated context files, and human context files. Before the knowledge cutoff, this corresponds to 17% of instances for SONNET-4.5, 40% of instances for GPT-5.2, and 12% of instances for GPT-5.1 MINI and QWEN3-30B-CODER. Figure 14 shows no significant or consistent difference between pre- and post-cutoff instance success rates, suggesting that contamination does not affect our conclusions.

C Validating CTXBENCH instances

In this section, we validate the quality of CTXBENCH. Specifically, we verify that developer-provided context files are significantly different from the LLM-generated ones. We also manually inspect CTXBENCH instances and validate the instance-description generation pipeline from CTXBENCH by applying it to SWE-BENCH.

Developer-provided context files differ from LLM-generated ones To compare developer-provided context files with LLM-generated ones, in Figure 15, we compare both their length distributions and their AI-detection scores. For the AI-detection score, we use the open-source Pangram

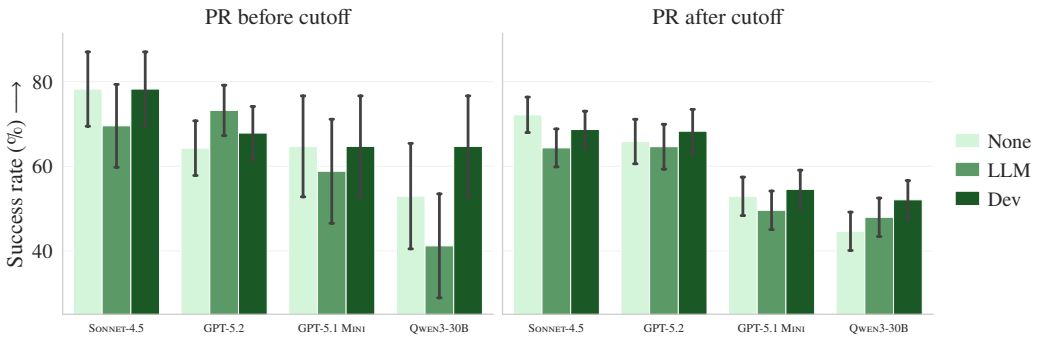


Figure 14: Resolution rate for four different models with LLM-generated context files and with developer-provided ones on CTXBENCH, split by knowledge cutoffs.

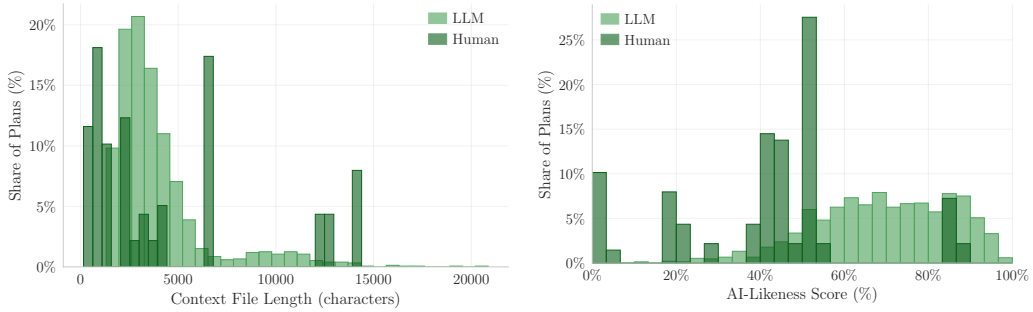


Figure 15: Length (left) and AI-detection score (right) distributions for LLM-generated and developer-provided context files on CTXBENCH.

LLM detection model EDITLENS LLAMA-3.2 3B [35]. If developer-provided context files were fully LLM-generated, they would follow distributions (both in length and AI-detection score) similar to those of the LLM-generated ones. Yet, we find in Figure 15 that developer-provided context files tend to be shorter than LLM-generated ones and, according to the Pangram detector, are significantly less likely to be AI-generated.

CTXBENCH instances are better specified than SWE-BENCH We manually inspect all CTXBENCH instances by examining the task descriptions and the corresponding unit tests. We find that most instances are well specified, with clear instructions on how to reproduce the issue and a technical description of the issue. For 25 of the 138 instances, we manually edit the test cases to remove over-specified tests. We provide an example of such a modification in Figure 16. In this case, the test still verifies that evaluating or parameterizing a library fails, but it no longer requires a particular error message string.

To confirm our inspection, we further generate a variant of SWE-BENCH in which we generate problem descriptions using the CTXBENCH pipeline (see §3), which enables us to evaluate the concrete impact of LLM-generated task descriptions compared to human baselines. We observe in Table 8 a consistent trend of about 15% higher accuracy, preserving all model rankings (with and without context files). We thus conclude that the CTXBENCH construction results in a slightly easier but fairer evaluation for coding agents compared to other benchmarks such as SWE-BENCH.

D Asset Licenses

In this section, we list the licenses of all assets used in this work.

Datasets

- SWE-BENCH LITE: MIT.

```
--- tests/pr/test_cli_lib.py (before)
+++ tests/pr/test_cli_lib.py (after)
@@ -45,7 +45,6 @@
 proc = run_cli("eval", "lib", str(prelude_path))

 assert proc.returncode != 0, "Evaluating a library should fail"
- assert "Can not evaluate a library" in proc.stderr


 def test_compile_lib_rejects_arguments() -> None:
@@ -53,4 +52,3 @@
 proc = run_cli("compile", "lib", str(prelude_path), "{}")

 assert proc.returncode != 0, "Supplying parameters to a library should fail"
- assert "Can not pass arguments to a library" in proc.stderr
```

Figure 16: Example manual edit to a generated CTXBENCH test for `opshin/opshin#171`.

Table 8: Resolution rate on SWE-BENCH for different models with either LLM-generated or original issue descriptions, under two context-file settings: no context files (None) and LLM-generated context files (LLM). Values in parentheses indicate the change relative to the corresponding no-context setting.

Model	No Context Files		LLM Context Files	
	LLM-generated Desc.	Original Desc.	LLM-generated Desc.	Original Desc.
GPT-5.2	70.8%	57.0%	66.8% (-4%)	54.9% (-2%)
QWEN3-30B	46.9%	30.3%	44.9% (-2%)	32.4% (+2%)

- CTXBENCH: all source repositories used to build the dataset have explicit open-source licenses that permit our use; the licenses are Apache-2.0, MIT, AGPL-3.0, GPL-3.0, and BSD-3-Clause.

Models and coding agents

- SONNET-4.5 and HAIKU-4.5: Anthropic Commercial Terms of Service.
- GPT-5.2 and GPT-5.1 MINI: OpenAI Services Terms.
- QWEN3-30B-CODER: Apache-2.0.
- GPT-OSS-120B: Apache-2.0.
- CODEX: Apache-2.0.
- QWEN CODE: Apache-2.0.
- CLAUDE CODE: Anthropic Commercial Terms of Service.

E Prompts

In this section, we detail all prompts used throughout this work.

E.1 CTXBENCH instances generation

We detail below the prompts used for filtering pull requests, setting up the instances, describing the instances, and generating the test cases.

Filtering pull requests

You are evaluating pull request `{pr_number}` for suitability as a regression-test task in SWE-bench style datasets. Decide whether the PR primarily introduces deterministic, testable behaviour. Such behaviors typically include bug fixes, but can also include feature additions as long as it is possible to write a precise specification that allows testing the new feature independently of the implementation.

Repository: `{repo_full_name}`

Title: `{title}`

Author: `{author}`

Merged at: `{merged_at}`

PR description

`{body}`

Diff excerpt

`{excerpt}`

Deliverables

1. Do **not** modify existing project code.
2. Create the JSON file `{decision_path}` with UTF-8 encoded content describing your decision using this schema:

```
{
  "pr_number": <int>,
  "suitable": <bool>,
  "needs_manual_review": <bool>,
  "decision": "include" | "exclude" | "manual_review",
  "rationale": "<short explanation>",
  "key_files": ["relative/file.py", "..."],
  "risk_factors": ["<short string>", "..."]
}
```

   - Set `"decision"` to `"include"` only when you are confident the PR is a self-contained bug fix that can be validated via regression tests.
   - Use `"manual_review"` if you are uncertain.
3. Stage the JSON file and finish. Do not stage anything else.

Setting up the instance

Your goal is to help developers set up their environment to run code in the repository and be able to run the current tests. You should write a list of all commands needed to (i) set up the environment from scratch, and (ii) run the existing tests. You need to make sure that the commands you provide actually work for you. The setup is considered valid if most of the tests are passing after running **exactly** your setup commands and the test commands you provide.

Test runner requirement

To run the repository tests, create a file at the root of the repository called `run_tests.py` that:

- executes all tests,
- parses the test output,
- writes a JSON file at the repository root named `test_results.json` with schema:

```
{"test_name": <bool>, ...}
```

where each `test_name` is the name of a test and the boolean indicates whether the test passed (`true`) or failed (`false`).

Deliverables

1. Create the JSON file `{decision_path}` with UTF-8 encoded content explaining the steps to set up the environment and run the tests (using the `run_tests.py` script you created):

```
{
  "setup_commands": ["<command1>", "<command2>", "..."],
  "test_commands": ["<command1>", "<command2>", "..."]
}
```

2. Create the script `run_tests.py` at the root of the repository.
3. Stage the JSON file and the script and finish. Do not stage anything else.

`{example_files_section}`

Describing the instance

You are given a pull request (PR) and the related issues for a given GitHub repository. Your goal is to format this information into a clear GitHub Issue following the template below.

- For the **Steps to Reproduce** field, only write the steps you actually took to reproduce the issue in your specific environment. Make those steps **reproducible and minimal**.
- Developers should be able to implement a solution similar to the one provided in the PR, but the Issue should **not leak the solution**.
- Save your output in Markdown format in the file `{metadata_relpath}`.

Feature requests: Specification required
Additionally, for issues about **adding a new feature** (rather than fixing a bug), include a **precise Specification** describing the desired behavior. It must be detailed enough to allow independent testing without relying on implementation details from the PR.

- Specify inputs (types, valid ranges, edge cases), outputs, side effects, and any required error handling.
- If the PR includes human-readable outputs (logs, UI text, error messages, ...), include them in the specification and state that fixes must use **exactly** those messages.

Issue template (copy into your Markdown output)

```
### Description
(Provide a clear and concise description of the problem.)

### Steps to Reproduce
1. [Step 1]
2. [Step 2]
3. ...

### Expected Behavior (if applicable)
(Explain what you expected to happen.)

### Actual Behavior (if applicable)
(Explain what actually happened.)

### Specification (if applicable)
(Provide a precise specification of the desired behavior.)

### Additional Information
(Add screenshots, logs, or other helpful details.)
```

Data for PR # `{pr_number}` **in repository** `{repo}` **at commit** `{commit_sha}`

PR description
`{pr_description}`

Referenced issues mentioned in the PR
`{referenced_issues_text}`

PR patch
`{pr_patch}`

PR test (if any)
`{pr_test_patch}`

Key files identified during triage
`{key_files_text}`

Generating the test cases

You are generating regression tests for pull request `{pr_number}` in `{repo}`. The current checkout is the base (pre-fix) commit `{commit_sha}`.

Problem description
`{problem_description}`

PR patch
`{pr_patch}`

PR test (if any)
`{pr_test_patch}`

Requirements

1. Focus on **deterministic** tests that expose the bug fixed by this PR. Tests should target **expected behavior** and must **not** rely on internal implementation details (variables, hidden helpers, etc.). They should fail on the base commit and pass on the merge commit

(after applying the PR patch). You must verify this property. You may apply the provided patch using `git apply`. If a specification is provided in the problem description, tests must **exactly** align with it. Avoid tests that depend on incidental choices (variable names, function names, strings, ...) unless explicitly required by the specification.

2. Create `run_pr_tests.py` at the repository root that executes *only* the tests you created, parses test output, and writes JSON results to `pr_test_results.json` with schema:

```
{"test_name": <bool>, ...}
```

   You may use `run_tests.py` as a reference. Note: your script should only run the tests you created for this PR.

3. Ensure new tests match the project's existing test style and conventions. First review existing tests to understand structure and framework. You may reuse tests from the PR if appropriate.
4. All new tests must be in **new files** created as part of this work. Do **not** modify any existing test files.
5. For `test_commands`, include any necessary steps (sourcing environments, setting variables, etc.) so tests run correctly in a fresh shell.

Deliverables

1. Create the new test files with your proposed tests.
2. Create the JSON file `{metadata_relpath}` with UTF-8 content explaining how to run the tests:

```
{
  "test_commands": ["<command1>", "<command2>", "..."],  # Commands to run the PR
     tests with `run_pr_tests.py`
  "test_files": ["path/to/test_file1", "path/to/test_file2", "..."]
}
```

3. Create the script `run_pr_tests.py` at the root of the repository.
4. Stage the JSON file and the script and finish. Do not stage anything else.

E.2 Analyzing Traces of Coding Agents

To analyze the tool calls made by the coding agents, we use GPT-OSS-120B with the prompt below.

Analyzing coding agent traces

You are labeling a tool call with a single intent category.

Goal: choose a category name that is:

- **Right-sized granularity:** more specific than "execute command" but not tied to exact arguments.
- **Reusable:** should apply to many future tool calls.
- **Clean:** do **not** include file paths, flags, quoted strings, IDs, repo names, or counts.

- **Format:** 2–5 words, lowercase, verb + object (e.g., `run tests`, `search codebase`). Avoid too-generic names like `run scripts`; specify what the script does (e.g., `compile code`).

You must also explain which tool is being used (e.g., `pytest`, `rg`, ...) in a dedicated field.

You will be given

- **tool_call:** the command or structured tool invocation
- **tool_output:** optional output text

Existing categories (use one if it fits): `{existing_tool_names}`

Decision rules

1. If one existing category fits, use it **exactly**.
2. If none fit, create **one** new category that:
   - is **not** tool-specific (avoid `pytest`, `kubectl`, `terraform`, etc.)
   - would likely match **5+** future tool calls
3. If the tool call does multiple things, pick the **primary intent** as the category (mention secondary intents in reasoning).

Return JSON only

```
{
  "tool_name": "<category>",
  "tool_used": "<specific tool or executable being invoked>",
  "reasoning": "<1-3 sentences: why this is the primary intent; include key clues from
     call/output; mention secondary intents if any>"
}
```

Tool call `{tool_call}`

Tool output (if any) `{tool_output}`