WIDESWE: CAN CODING AGENTS COORDINATE CHANGES ACROSS REPOSITORIES?

Anonymous authors
Paper under double-blind review

ABSTRACT

Coding-agent evaluation has progressed from resolving individual issues to carrying out long-horizon development, yet task completion is still largely assessed within a single codebase. In software ecosystems, many features and bug fixes require coordinated changes across multiple repositories. We introduce WIDESWE to evaluate coding agents on such cross-repository tasks. Mining and reviewing changes across 103 software ecosystems yields 120 real-world tasks, balanced between 60 bug fixes and 60 features. We derive prompts from related issues and pull requests. We systematically review and adapt hidden tests to support diverse correct implementations while preserving required behavior and regression checks. Across seven agent configurations, full task success ranges from 10.83% to 42.50%, with the configuration pairing Codex CLI with GPT-5.6-sol achieving the highest rate. Trajectories show agents failing to identify necessary changes, recognizing changes but leaving them unfinished, or modifying the required repositories without fully satisfying the request. To examine whether working on one repository at a time can alleviate these difficulties, we compare it with joint execution under identical prompts. Independent execution mainly recovers omitted work and is less effective at correcting previously attempted but unsuccessful implementations. Joint execution can use information from related repositories to guide implementation and verification. Code is available at https://anonymous.4open.science/r/WIDESWE-08BB.

1 INTRODUCTION

Large language models (LLMs) now serve as the foundation of coding agents that can inspect repositories, edit multiple files, run tests, and iteratively carry out software-engineering tasks (Liu et al., 2024a; Wang et al., 2025c). Evaluation has expanded beyond isolated code generation (Chen et al., 2021; Austin et al., 2021; Wang et al., 2025d; Zhang et al., 2026b) toward autonomous work in real repositories (Pan et al., 2024). SWE-bench asks agents to resolve real GitHub issues in a repository (Jimenez et al., 2024). More recent benchmarks extend this scope: DeepSWE evaluates original, long-horizon engineering tasks (Huang et al., 2026), while ProgramBench requires rebuilding complete programs from reference executables and documentation (Yang et al., 2026b). Despite this broader scope, these benchmarks still focus primarily on completing tasks in a single repository.

In software ecosystems, however, a single feature or bug fix may require coordinated changes across several repositories (Blincoe et al., 2019; Ma et al., 2017). For example, introducing a shared feature across Sentry's language SDKs[1] requires separate implementations in its Go, Python, and Ruby repositories. Other changes involve dependencies: a new capability in Sentry's PHP SDK also requires corresponding updates to two other repositories that depend on it. Cross-repository links are common in practice: our analysis of 1,729,171 PRs across 103 software ecosystems identified 109,233 PR records explicitly referencing another repository in the same ecosystem. Can coding agents handle this coordination and complete the request across repositories?

To examine this question, we first evaluate an agent configuration pairing Codex CLI with GPT-5.6-sol. As Figure 1 illustrates, failures occurred in identifying the full scope of affected repositories, delivering the required changes in each, and satisfying task requirements after editing. In Sentry,

[1] https://github.com/getsentry

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.

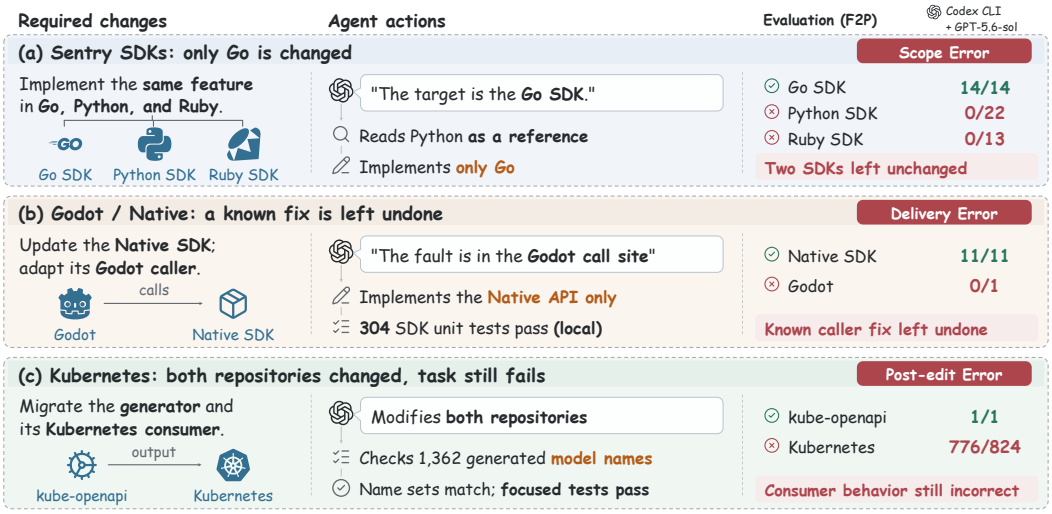


Figure 1: Three observed cross-repository failures in Codex CLI–GPT-5.6-sol runs.

the agent narrows a three-SDK requirement to Go, leaving Python and Ruby unchanged. In the Godot/Native task, both repositories require changes; the agent recognizes this scope but modifies only the Native SDK. In Kubernetes, it identifies and modifies both required repositories, but the changes are incorrect and the task remains unsolved.

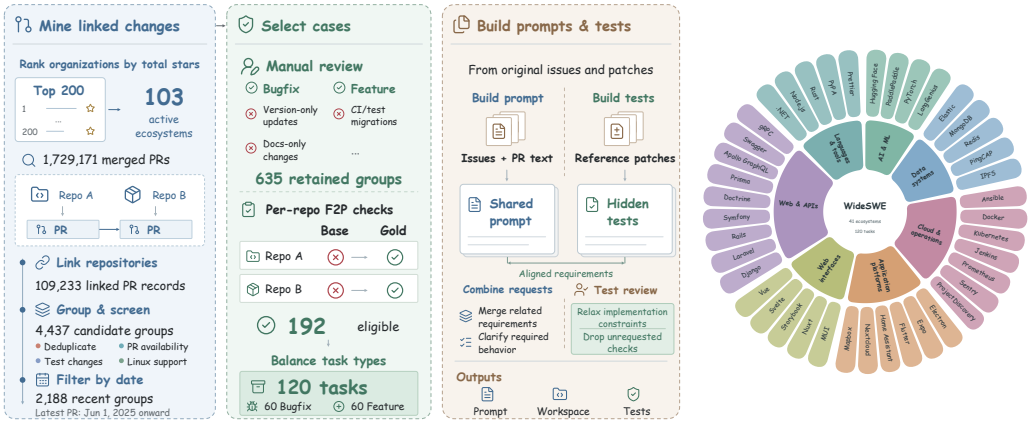
To assess agents' ability to handle these challenges, we build WIDESWE, grouping changes that implement the same feature or fix across repositories into a single task and evaluating the request as a whole. From the 200 GitHub organizations ranked highest by total repository stars,[2] we identify 103 active software ecosystems whose repositories support a common product, platform, or technology. We mine linked, merged PRs and use manual review and executable validation to identify tasks requiring substantive changes in multiple repositories. Balancing task types yields 120 cases: 60 bug fixes and 60 features. We merge requirements from the original issues and PRs into task prompts. We also address mismatches between PR requirements and inherited tests, preserving required behavior while supporting alternative correct implementations.

Our experiments show that agents do not reliably carry a shared requirement through all affected repositories. Codex CLI–GPT-5.6-sol solves only 42.50% of cases. In 48 of its 69 failed cases, at least one repository passes all its fail-to-pass (F2P) tests while another does not. The trajectories show the agent missing part of the required scope or stopping after tests pass in one repository, leaving already identified downstream work unfinished. We further compare joint execution in the ecosystem workspace with independent execution in each target repository. Across 89 cases with identical prompts, joint execution solves 36 and independent execution solves 32, with 20 outcome reversals. Trajectory analysis reveals two contrasting patterns: independent execution can recover omitted work by narrowing the scope, but can also lose behavioral references from related repositories.

Our contributions are threefold:

- **A task formulation and benchmark for cross-repository coding.** We formulate cross-repository task completion as implementing one shared feature or bug fix across multiple target repositories. We introduce WIDESWE, comprising 120 real-world tasks across 41 software ecosystems.
- **Requirement-aligned test review.** We identify mismatches between task requirements and inherited tests, and address them through rule-guided manual revisions that preserve required functionality while supporting alternative correct implementations.
- **An empirical study of cross-repository task completion.** We evaluate seven agent configurations, examining differences across models, scaffolds, task types, repository counts, and language diversity. We combine joint-versus-independent comparisons with trajectory analysis to provide empirical insights into cross-repository task completion.

[2]Gitstar Ranking: https://gitstar-ranking.com/organizations, collected on June 5, 2026.



(a) Mining and case selection (b) Ecosystem coverage

Figure 2: WIDESWE construction and coverage. (a) Mining and case selection. (b) The 41 ecosystems grouped by application area.

2 THE WIDESWE BENCHMARK

2.1 TASK FORMULATION

Each WIDESWE task represents one feature or bug fix requiring coordinated changes in at least two *target repositories*. Coordination involves not only handling explicit interface dependencies but also autonomously identifying which repositories require changes to fulfill the shared request and completing those changes while preserving existing behavior and cross-repository consistency.

For task c, let p_c denote the request and $\mathcal{W}_c$ the historical workspace containing repositories $\mathcal{R}_c$. The target set $\mathcal{T}_c \subseteq \mathcal{R}_c$, with $|\mathcal{T}_c| \geq 2$, comprises the repositories that require substantive changes to fulfill the request. Repositories in $\mathcal{R}_c \setminus \mathcal{T}_c$ provide additional ecosystem context. In addition to the targets, agents can access up to 20 context repositories in a task workspace (Figure 6 in Appendix A.3).

Given p_c and $\mathcal{W}_c$, an agent A can inspect and modify the workspace and produces a collection of repository patches Δ_c:

$$\Delta_c = A(p_c, \mathcal{W}_c), \qquad \mathcal{W}'_c = \mathcal{W}_c \oplus \Delta_c, \tag{1}$$

where $\oplus$ applies the patches to the original snapshots and $\mathcal{W}'_c$ is the resulting workspace. Target repositories determine the scoring scope; context repositories remain available for inspection and modification. Reference patches and hidden tests are withheld from the agent.

2.2 BENCHMARK CONSTRUCTION

Ecosystem and Change Mining From the top 200 GitHub organizations ranked by total repository stars (Gitstar Ranking, June 5, 2026), we select 103 active software ecosystems supporting a common product, platform, or technology. We collect PRs merged since January 1, 2024 in non-archived repositories and identify explicit links to other repositories within the same ecosystem. Among 1,729,171 PRs, 109,233 contain such links. We group linked changes across repositories into candidates. After deduplication and checks for PR availability, test-related changes, and Linux compatibility, 4,437 candidate groups remain.

Case Selection and Composition Figure 2(a) summarizes the selection process. We focus on 2,188 candidate groups whose latest PR was merged on or after June 1, 2025. Manual review retains 635 groups that address a single feature or bug fix and require substantive changes across repositories. Executable validation and final review yield 192 eligible cases, each with at least one F2P test per target repository that fails before the reference changes and passes afterward. To balance bug fixes and features, we retain all 60 bug-fix cases and select 60 of the 132 feature cases, considering the

Table 1: Main results by task type. Codex denotes Codex CLI, and CC denotes Claude Code. Model headers abbreviate GPT-5.6-sol, Claude Opus 5, Gemini 3.8 Flash, DeepSeek V4 Pro, and Qwen 3.8 Max. Case/Repo denote task/repository success (%); F2P/P2P denote F2P-complete/P2P-preserved repositories (%). API denotes mean API requests per task. Bold marks the highest rate in each row.

Task	Metric	GPT-5.6 + Codex	GPT-5.6 + CC	Opus 5 + CC	Gemini + CC	DeepSeek + CC	Qwen + CC	GLM 5.3 + CC
	Case ↑	41.67	33.33	41.67	21.67	38.33	**48.33**	30.00
	Repo ↑	61.67	55.83	63.33	26.67	60.00	**66.67**	52.50
Bugfix	F2P ↑	62.50	58.33	64.17	28.33	62.50	**67.50**	52.50
	P2P ↑	95.37	91.67	**96.30**	**96.30**	92.59	**96.30**	89.81
	API ↓	62.8	76.9	118.7	178.4	117.3	141.5	113.9
	Case ↑	**43.33**	31.67	28.33	0.00	15.00	26.67	10.00
	Repo ↑	**65.41**	57.89	48.12	6.02	36.84	48.12	28.57
Feature	F2P ↑	**68.42**	61.65	48.87	6.02	39.10	48.12	30.83
	P2P ↑	86.73	78.76	85.84	**95.58**	77.88	88.50	78.76
	API ↓	110.7	191.4	201.8	113.9	203.7	212.7	188.2
	Case ↑	**42.50**	32.50	35.00	10.83	26.67	37.50	20.00
	Repo ↑	**63.64**	56.92	55.34	15.81	47.83	56.92	39.92
Overall	F2P ↑	**65.61**	60.08	56.13	16.60	50.20	57.31	41.11
	P2P ↑	90.95	85.07	90.95	**95.93**	85.07	92.31	84.16
	API ↓	86.7	134.8	160.2	146.1	160.2	177.1	151.1

diversity of required behaviors. The resulting 120 tasks cover 41 ecosystems (Figure 2(b)). Detailed selection criteria and dataset distributions appear in Appendix A.

Prompt Construction We construct each prompt from the original issues and PR descriptions, preserving their wording wherever possible and combining related requirements into one cross-repository task. We clarify necessary behavior and edge cases without introducing additional implementation instructions. Appendix A.5 illustrates this process with a comparison between the original text and the final prompt.

Hidden-Test Construction and Review We construct hidden tests by extracting test changes from the reference patches. During review, we find that some tests impose implementation-specific constraints beyond the task requirements, rejecting otherwise correct solutions. DeepSWE reports similar findings (Huang et al., 2026). We address these mismatches using two review rules, guided by the original issues, PR descriptions, and task prompts:

1. **Relax implementation-specific constraints.** Some tests require specific private helper names, files at fixed paths, or exact text in error messages, even though the task does not require these choices. We remove these restrictions while preserving required behavior and existing interface contracts.
2. **Remove unrequested functionality checks.** Some reference patches introduce additional functionality required by neither the original issue/PR nor the task prompt. We remove tests for these additions while retaining checks for required functionality and existing behavior.

For example, a TensorDict test checks that an invalid argument raises `TypeError` and that the error message contains "generator must be." We retain the exception-type check but remove the message-text restriction. Appendix A.8 illustrates both review rules. We verify that the revised test suites still reject the original code and accept the reference solutions.

2.3 Evaluation

We evaluate the patched workspace $\mathcal{W}'_c$ using hidden tests supplied independently of agent edits (Appendix A.7). For each target repository $r \in \mathcal{T}_c$, let $F_{c,r}$ denote the fail-to-pass (F2P) tests that fail on the original code and pass with the reference changes, and let $P_{c,r}$ denote the pass-to-pass (P2P) regression tests that pass in both states.

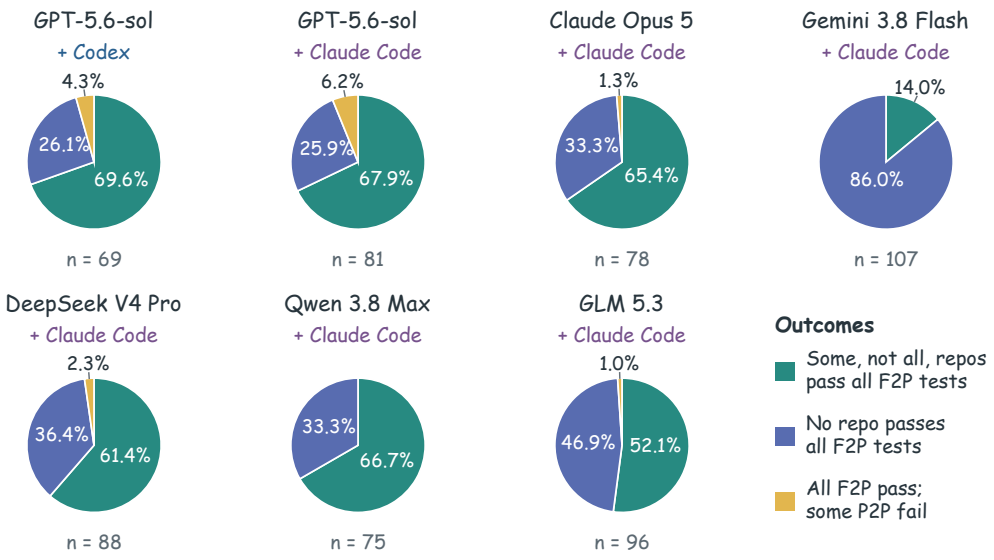


Figure 3: Repository completion within unresolved tasks; n is the number of unresolved tasks. Colors denote mutually exclusive outcomes.

Define $y_{c,r}(\mathcal{W}'_c)$ as 1 if every test in $F_{c,r} \cup P_{c,r}$ passes with no missing or invalid results, and 0 otherwise. A task is solved only when every target repository succeeds:

$$\text{Success}(c) = \prod_{r \in \mathcal{T}_c} y_{c,r}(\mathcal{W}'_c). \tag{2}$$

The task success rate averages $\text{Success}(c)$ over evaluated tasks. Repository-level and test-level results provide additional diagnostics.

3 Experimental Setup

We evaluate seven agent configurations, each pairing a coding-agent scaffold, an LLM, and a reasoning-effort setting. The scaffolds are Codex CLI (v0.147.0) and Claude Code (v2.1.139). Codex CLI is paired with GPT-5.6-sol (high), while Claude Code is paired with GPT-5.6-sol (high), Claude Opus 5 (high), Gemini 3.8 Flash (xhigh), DeepSeek V4 Pro (high), Qwen 3.8 Max (Qwen Team, 2026) (xhigh), and GLM 5.3 (xhigh). For each task, the agent receives one prompt and a historical ecosystem workspace, where it can inspect and modify all repositories. All configurations use the same harness-level permissions, timeout policy, and per-run resource limits.

We report results separately for bug fixes, features, and all tasks combined. *Case-level success* is the primary metric and requires all target repositories to pass all F2P and P2P tests. *Repository-level success* applies the same criterion to each target repository. *F2P-complete* and *P2P-preserved* report the percentage of repositories passing all corresponding tests, among those with such tests. Test-level pass rates appear in Appendix B.2. API measures mean requests per task with recorded usage.

We ask:

RQ1 How effectively can coding agents complete cross-repository tasks?
RQ2 How does success vary with task type, repository count, and language diversity?
RQ3 How does joint execution compare with independent execution in each target repository under identical prompts?

For RQ3, we compare joint and independent execution using the Codex CLI–GPT-5.6-sol configuration on 89 cases (29 bug fixes and 60 features) whose prompts apply unchanged in both settings (Appendix B.1). The other 31 cases require repository-specific prompt adaptations and are excluded to keep the prompts identical. Selected trajectories in Appendix C complement the quantitative results with comparisons across execution settings, agent scaffolds, and models.

Table 2: Failure categories (%).

Configuration	Scope	Delivery	Post-edit
GPT + Codex	37.68	2.90	59.42
GPT + CC	30.86	1.23	67.90
Opus + CC	25.64	6.41	67.95
Gemini + CC	37.38	**52.34**	10.28
DeepSeek + CC	30.68	7.95	61.36
Qwen + CC	21.33	5.33	73.33
GLM 5.3 + CC	30.21	9.38	60.42

Table 3: Joint versus Independent execution.

Group	Metric	Joint	Independent
Task success (%)	Bugfix	34.48	**44.83**
	Feature	**43.33**	31.67
	Overall	**40.45**	35.96
Repository outcomes (%)	Repo	**61.78**	60.73
	F2P	**64.40**	**64.40**
	P2P	**89.63**	85.37
	API	**94.7**	296.6

4 RESULTS

4.1 RQ1: PERFORMANCE ON CROSS-REPOSITORY TASKS

Table 1 summarizes results on 120 tasks spanning 253 target repositories, with 2,815 F2P and 22,139 P2P tests. The best-performing configuration, Codex CLI–GPT-5.6-sol, fully solves only 42.50% of tasks; the other configurations achieve 10.83%–37.50%. Yet Codex CLI–GPT-5.6-sol solves at least one target repository in 83.33% of tasks, revealing a gap between repository-level progress and complete task resolution. We examine the unresolved tasks to understand where work remains incomplete and how these gaps arise.

Functional completion within unresolved tasks. Figure 3 examines the unresolved tasks. For every configuration except Gemini, most of these tasks (52.08%–69.57%) have at least one repository passing all F2P tests, but not all repositories do so. For Gemini, 85.98% have no repository passing all F2P tests. Failures due solely to P2P tests are uncommon. Together with the high P2P preservation rates in Table 1, these results indicate that agents generally preserve existing tested behavior but often leave the requested functionality incomplete.

These results do not explain why agents leave work incomplete. We classify unresolved runs using a two-stage procedure based on final repository diffs and trajectory evidence; Appendix B.3 details the annotation rules and inter-annotator agreement. We group failures into the following three categories.

Trajectory analysis locates where completion breaks down. Table 2 groups unresolved runs into three categories. *(1) Incomplete scope identification:* the agent leaves a necessary repository change unrecognized, or incorrectly decides that the repository needs no changes even after inspecting it. *(2) Recognized work without delivery:* the agent identifies a necessary change or diagnoses the defect requiring it, but does not deliver the change. *(3) Post-edit failure:* all target repositories receive substantive code changes, but the run still fails the required behavior or regression checks.

For the Codex CLI–GPT-5.6-sol configuration, the shares in Table 2 are 37.68%, 2.90%, and 59.42%, respectively. Post-edit failure is most common. Some runs fully fix one repository but make only part of the necessary changes in another, leaving the problem unresolved. Others connect a caller to new functionality in another repository but change the caller's existing behavior, causing regression tests to fail. A subtler failure occurs in Sentry/Symbolicator: the agent makes the sender and receiver use an exception-name string, and its tests also use strings, although the task requires structured exception information. Both implementations and the added tests share the same incorrect assumption, allowing local tests to pass without delivering the required behavior (Appendix C.1).

Some Delivery failure cases show that even with a clear task scope, agents may still fail to complete the implementation after many API calls. For example, DeepSeek makes 769 API calls on the Prettier/yaml-unist-parser task, explicitly planning changes to both repositories early on but continuing to work on only one, ultimately leaving Prettier unchanged. Gemini makes 562 API calls on the Graphon/Dify task, completing Graphon before moving on to investigate Dify but never delivering the Dify changes.

Same scaffold, different models. With Claude Code fixed, Table 2 shows distinct patterns among each model's unresolved runs. In 52.34% of Gemini's failures, it identifies the necessary work but leaves changes undelivered; another 37.38% involve incomplete scope identification. Within the

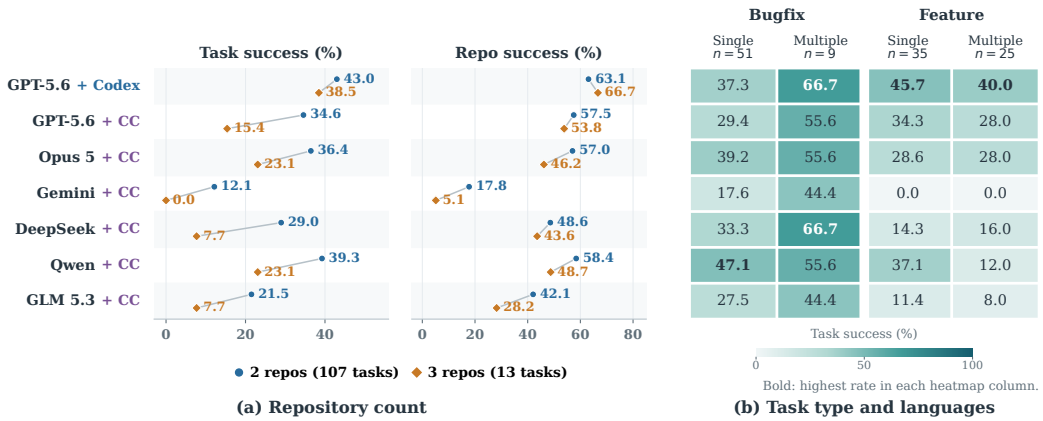


Figure 4: (a) Task and repository success by target-repository count. Connected points compare separate task groups. (b) Task success by intent and language diversity; n denotes the number of tasks. Language families describe reference changes. All rates are percentages.

former category, 82.14% remain in investigation, solution planning, or preparatory checks rather than implementing the requested behavior; 17.86% begin implementing in some repositories but leave recognized work in others undone. In these failed runs, Gemini often remains in analysis rather than proactively implementing the required changes. In Elastic Agent/Fleet Server, for example, Gemini explicitly plans client-side request compression and server-side support, but continues investigating configuration and request handling without implementing either change (Appendix C.4, Figure 20).

Recognized work without delivery is less common for DeepSeek (7.95%), Qwen (5.33%), and GLM (9.38%): their omissions more often involve failing to recognize the full modification scope. Post-edit failures dominate for Opus (67.95%) and Qwen (73.33%), as well as GLM. Unlike Gemini, which frequently leaves recognized work unfinished, these models often locate and modify the relevant repositories, yet still struggle to make those changes jointly satisfy the request. Appendix C.4 presents selected comparative trajectories.

Same model, different scaffolds. Table 1 shows that, with GPT-5.6-sol fixed, moving from the Codex CLI scaffold to Claude Code reduces task success from 42.50% to 32.50%, while mean recorded API requests per task rise from 86.7 to 134.8. The Codex CLI pairing is more effective in this evaluation; the higher request count does not correspond to higher task completion. Appendix C.3 provides a case comparison of the same task under the two scaffolds.

4.2 RQ2: Task Intent, Repository Scope, and Language Diversity

We compare task type, language diversity, and repository count using Table 1 and Figure 4, then examine failure behaviors. Language families describe reference changes (Appendix A.4).

Model performance varies by task type and language mix. The Claude Code–Qwen configuration outperforms the Codex CLI–GPT-5.6-sol configuration on bug fixes but falls behind on features (Table 1). Figure 4(b) further shows that all seven configurations achieve higher success on multiple-family bug fixes than on single-family bug fixes, but feature tasks do not follow the same pattern. For example, Qwen leads the Claude Code configurations on single-family features at 37.14%, but falls to 12.00% on multiple-family features, below GPT and Opus at 28.00%. Opus remains comparatively stable across the two feature groups. Language count alone is therefore insufficient to judge task difficulty; task type and the specific changes required also matter.

Performance by Repository Count. In our experiments, all configurations have lower task success on three-repository tasks (Figure 4(a)). This gap does not necessarily arise from overlooking more repositories. For the Codex CLI–GPT-5.6-sol configuration, repository success rises from 63.08% on two-repository tasks to 66.67% on three-repository tasks, while task success falls from 42.99% to 38.46%. Among its failed three-repository tasks, 87.50% modify all targets and 62.50% solve two of

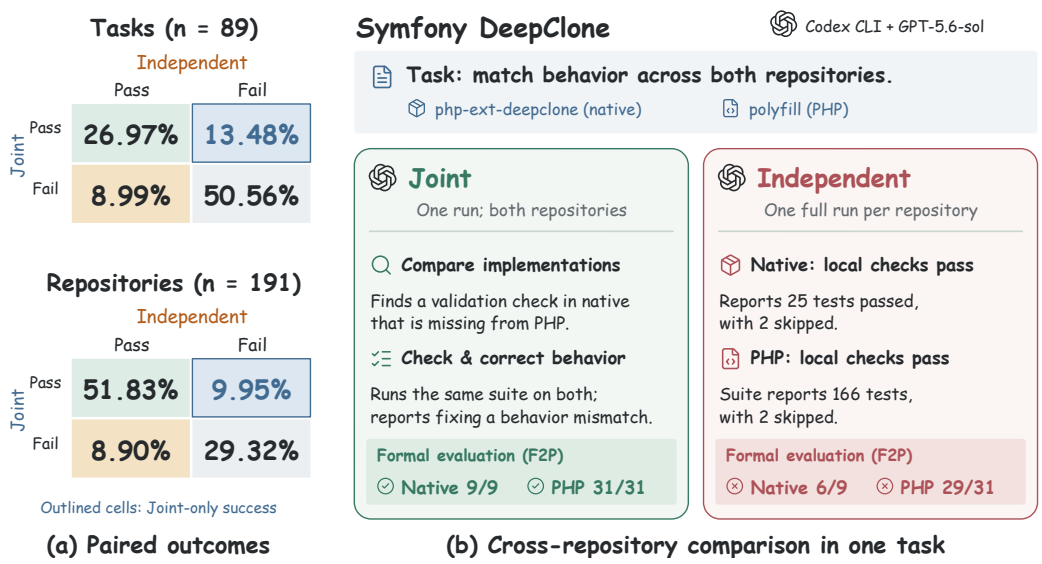


Figure 5: (a) Paired task and repository outcomes (%); off-diagonal cells show success reversals. (b) A Symfony task where Joint compares related implementations and both repositories pass, while Independent leaves failures in both. F2P scores show passed/total hidden checks, not local tests.

them. These results suggest that this configuration usually covers the repositories requiring changes, but coordinating the required adaptations may become more difficult as the number of repositories increases. A three-repository Sentry task illustrates this distinction in a Claude Code–DeepSeek run: the agent modifies all three targets and completes the PHP SDK and Laravel changes, but adds an integer-only configuration node in Symfony, rejecting the `null` setting explicitly allowed by the request (Appendix C.1, Figure 15).

4.3 RQ3: Joint versus Independent Execution

Independent execution does not consistently improve task success. Table 3 compares joint and independent execution of the Codex CLI–GPT-5.6-sol configuration under identical prompts on 89 tasks. Joint execution solves 40.45%, versus 35.96% when every independent repository run must succeed. Yet 22.47% of tasks change outcome in either direction (Figure 5). Independent execution improves bug-fix success from 34.48% to 44.83%, but reduces feature success from 43.33% to 31.67%. This contrast suggests that narrowing the work scope affects bug fixes and features differently. For bug fixes, a narrower scope can help agents focus on locating and repairing defects in the current repository. Features often require repositories to jointly implement new behavior; separating them may remove context needed to understand interface contracts and compare related implementations, making coordinated implementation harder.

Independent execution uses 296.6 API requests per task on average, compared with 94.7 for Joint, because each target receives a separate full run. Despite using 3.13 times as many requests, independent execution does not improve overall task success.

Independent execution reduces each run's scope and workload. Among repositories that fail in Joint execution, 60.00% of those left unmodified succeed independently, compared with only 9.43% of those already modified (Table 6 in Appendix B.1). This suggests that a separate run offers much less improvement when an implementation has already been attempted but remains incorrect. In the Sentry Go/Python/Ruby task, the Joint run completes only the Go SDK, whereas separate runs complete all three SDKs under the same prompt (Appendix C.2, Figure 16). Additionally, in the MUI Strict Mode bug-fix task, the Joint run recognizes that both repositories require changes but ends after repairing Base UI and resolving its testing setup, forgetting to modify Material UI. Separate runs repair both repositories. This case suggests that handling a larger workload within a joint run may lead the agent to forget required changes in another repository.

Cross-repository context supports both implementation and verification. Figure 5 shows that 13.48% of tasks and 9.95% of repositories succeed only in Joint execution. Trajectories illustrate two uses of cross-repository context. First, it supplies facts needed for implementation. In Ansible, the Joint run uses one repository's implementation to guide compatible changes in the other. The independent run instead makes incorrect assumptions about the other repository, producing incompatible code (Appendix C.2, Figure 17). Second, it provides behavioral references for verification. In Symfony, the two repositories must implement the same behavior. The Joint run compares their implementations and corrects a difference, allowing both repositories to pass (Figure 5(b)); the independent runs leave errors unresolved in both (Appendix C.2, Figure 18).

5 Related Work

Coordination in software ecosystems. Dependencies between software projects extend development work beyond a single project (Blincoe et al., 2015; Decan et al., 2019). Reference Coupling identifies technical dependencies through cross-project references in development records (Blincoe et al., 2019). Ma et al. (2017) study how developers trace bug causes across projects and coordinate upstream and downstream repairs. An industrial study further documents how teams maintaining interdependent components coordinate interface changes, work progress, and priorities (Begel, 2008). These studies examine project dependencies and human coordination; WideSWE turns real cross-repository changes into executable tasks, evaluating whether coding agents can identify the required change scope and fulfill a shared request across repositories.

Software-engineering agents and benchmarks. RepoBench and CrossCodeEval (Liu et al., 2024b; Ding et al., 2023) evaluate code completion (Hindle et al., 2016; Raychev et al., 2014) using cross-file repository context (Zhang et al., 2023; Wu et al., 2024; Le Hai et al., 2025; Wang et al., 2025a; Zhao et al., 2025; Wang et al., 2026). SWE-bench evaluates real GitHub issue resolution through executable tests (Jimenez et al., 2024), a setting studied by SWE-agent, Agentless, AutoCodeRover, and OpenHands (Yang et al., 2024a; Xia et al., 2025; Zhang et al., 2024; Wang et al., 2025b). Subsequent benchmarks broaden evaluation along several dimensions (Yang et al., 2024b; Zhang et al., 2026a; Badertdinov et al., 2026a;b). Multi-SWE-bench, SWE-PolyBench, and SWE-bench Multilingual extend language coverage beyond Python (Zan et al., 2026; Rashid et al., 2025; Yang et al., 2026a). SWE-Bench Pro and SWE-EVO emphasize long-horizon engineering and software evolution (Deng et al., 2025; Le et al., 2025). DeepSWE introduces original long-horizon tasks with functional verifiers, while ProgramBench requires reconstructing programs from executables and documentation (Huang et al., 2026; Yang et al., 2026b). Broader language coverage and longer tasks expand agents' work within a project; they do not themselves evaluate whether agents can complete coordinated changes across repositories.

Cross-repository evaluation. BeyondSWE's CrossRepo tasks require agents to resolve an issue in a target repository using code and solutions from external repositories (Chen et al., 2026). WideSWE focuses on implementing one shared feature or fix across multiple scored target repositories. Related repositories can provide useful context, but they also contain required changes that must be delivered and verified. Task success therefore requires all target repositories to pass their required tests.

6 Conclusion

We introduce WideSWE, a benchmark of 120 real-world tasks requiring coordinated changes across repositories to implement one feature or fix. Across the seven evaluated configurations, the highest task success rate is 42.50%. Agents miss necessary changes, leave recognized work unfinished, or modify the required repositories without fully satisfying the request. Independent execution can recover omitted work but does not improve overall task success in our paired comparison. Joint execution can provide information from related repositories that helps agents implement and verify changes. WideSWE shifts the focus from progress in individual repositories to whether changes across repositories jointly fulfill the task requirements.

AI USE STATEMENT

We used generative AI tools to assist with manuscript writing and editing, code and test development, and result analysis. The authors reviewed the AI-assisted work and are responsible for the content.

ETHICS STATEMENT

The benchmark is constructed from public open-source repositories and public development records. The released artifact preserves licenses and attribution, excludes credentials and private data, and documents responsible use. Agent-generated patches may contain vulnerabilities or regressions and should not be deployed without human review.

REPRODUCIBILITY STATEMENT

The artifact includes instance metadata, repository and base-commit identifiers, provenance records, prompts, hidden-test patches, environment definitions, scoring manifests, agent patches where licensing permits, and scripts for reproducing aggregate metrics. Section 2 describes construction and evaluation, and Section 3 defines the experimental conditions and metrics.

REFERENCES

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021.

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. *Advances in Neural Information Processing Systems*, 38, 2026a.

Ibragim Badertdinov, Maksim Nekrashevich, Anton Shevtsov, and Alexander Golubev. Swe-rebench v2: Language-agnostic swe task collection at scale. *arXiv preprint arXiv:2602.23866*, 2026b.

Andrew Begel. Effecting change: Coordination in large-scale software development. In *Proceedings of the 2008 international workshop on Cooperative and human aspects of software engineering*, pp. 17–20, 2008.

Kelly Blincoe, Francis Harrison, and Daniela Damian. Ecosystems in github and a method for ecosystem identification using reference coupling. In *2015 IEEE/ACM 12th working conference on mining software repositories*, pp. 202–211. IEEE, 2015.

Kelly Blincoe, Francis Harrison, Navpreet Kaur, and Daniela Damian. Reference coupling: An exploration of inter-project technical dependencies and their characteristics within large software ecosystems. *Information and Software Technology*, 110:174–189, 2019.

Guoxin Chen, Fanzhe Meng, Jiale Zhao, Minghao Li, Daixuan Cheng, Huatong Song, Jie Chen, Yuzhi Lin, Hui Chen, Xin Zhao, et al. Beyondswe: Can current code agent survive beyond single-repo bug fixing? *arXiv preprint arXiv:2603.03194*, 2026.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

Alexandre Decan, Tom Mens, and Philippe Grosjean. An empirical comparison of dependency network evolution in seven software packaging ecosystems. *Empirical Software Engineering*, 24 (1):381–416, 2019.

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*, 2025.

Yangruibo Ding, Zijian Wang, Wasi Ahmad, Hantian Ding, Ming Tan, Nihal Jain, Murali Krishna Ramanathan, Ramesh Nallapati, Parminder Bhatia, Dan Roth, et al. Crosscodeeval: A diverse and multilingual benchmark for cross-file code completion. *Advances in Neural Information Processing Systems*, 36:46701–46723, 2023.

Abram Hindle, Earl T Barr, Mark Gabel, Zhendong Su, and Premkumar Devanbu. On the naturalness of software. *Communications of the ACM*, 59(5):122–131, 2016.

Wenqi Huang, Charley Lee, Leonard Tng, and Serena Ge. Deepswe: Measuring frontier coding agents on original, long-horizon engineering tasks. *arXiv preprint arXiv:2607.07946*, 2026.

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*, volume 2024, pp. 54107–54157, 2024.

Tue Le, Minh VT Thai, Dung Nguyen Manh, Huy Phan Nhat, and Nghi DQ Bui. Swe-evo: Benchmarking coding agents in long-horizon software evolution scenarios. *arXiv preprint arXiv:2512.18470*, 2025.

Nam Le Hai, Dung Manh Nguyen, and Nghi DQ Bui. On the impacts of contexts on repository-level code generation. In *Findings of the Association for Computational Linguistics: NAACL 2025*, pp. 1496–1524, 2025.

Jiawei Liu, Jia Le Tian, Vijay Daita, Yuxiang Wei, Yifeng Ding, Yuhan Katherine Wang, Jun Yang, and Lingming Zhang. Repoqa: Evaluating long context code understanding. *arXiv preprint arXiv:2406.06025*, 2024a.

Tianyang Liu, Canwen Xu, and Julian McAuley. Repobench: Benchmarking repository-level code auto-completion systems. In *International Conference on Learning Representations*, volume 2024, pp. 47832–47850, 2024b.

Wanwangying Ma, Lin Chen, Xiangyu Zhang, Yuming Zhou, and Baowen Xu. How do developers fix cross-project correlated bugs? a case study on the github scientific python ecosystem. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pp. 381–392. IEEE, 2017.

Jiayi Pan, Xingyao Wang, Graham Neubig, Navdeep Jaitly, Heng Ji, Alane Suhr, and Yizhe Zhang. Training software engineering agents and verifiers with swe-gym. *arXiv preprint arXiv:2412.21139*, 2024.

Qwen Team. Qwen3.8-max: A new bar for coding and cowork, August 2026. URL https://qwen.ai/blog?id=qwen3.8.

Muhammad Shihab Rashid, Christian Bock, Yuan Zhuang, Alexander Buchholz, Tim Esler, Simon Valentin, Luca Franceschi, Martin Wistuba, Prabhu Teja Sivaprasad, Woo Jung Kim, et al. Swe-polybench: A multi-language benchmark for repository level evaluation of coding agents. *arXiv preprint arXiv:2504.08703*, 2025.

Veselin Raychev, Martin Vechev, and Eran Yahav. Code completion with statistical language models. In *Proceedings of the 35th ACM SIGPLAN conference on programming language design and implementation*, pp. 419–428, 2014.

Baoyi Wang, Xingliang Wang, Guochang Li, Chen Zhi, Junxiao Han, Xinkui Zhao, Nan Wang, Shuiguang Deng, and Jianwei Yin. Greprag: An empirical study and optimization of grep-like retrieval for code completion. *arXiv preprint arXiv:2601.23254*, 2026.

Xingliang Wang, Baoyi Wang, Chen Zhi, Junxiao Han, Xinkui Zhao, Jianwei Yin, and Shuiguang Deng. Grace: Graph-guided repository-aware code completion through hierarchical code fusion. *arXiv preprint arXiv:2509.05980*, 2025a.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, Hoang Tran, Fuqiang Li, Ren Ma, Mingzhang Zheng, Bill Qian, Daniel Shao, Niklas Muennighoff, Yizhe Zhang, Binyuan Hui, Junyang Lin, Robert Brennan, Hao Peng, Heng Ji, and Graham Neubig. Openhands: An open platform for ai software developers as generalist agents. In Y. Yue, A. Garg, N. Peng, F. Sha, and R. Yu (eds.), *International Conference on Learning Representations*, volume 2025, pp. 65882–65919, 2025b. URL https://proceedings.iclr.cc/paper_files/paper/2025/file/a4b6ad6b48850c0c331d1259fc66a69c-Paper-Conference.pdf.

Yunkun Wang, Yue Zhang, Guochang Li, Chen Zhi, Binhua Li, Fei Huang, Yongbin Li, and Shuiguang Deng. Inspectcoder: Dynamic analysis-enabled self repair through interactive llm-debugger collaboration. *arXiv preprint arXiv:2510.18327*, 2025c.

Yunkun Wang, Yue Zhang, Zhen Qin, Chen Zhi, Binhua Li, Fei Huang, Yongbin Li, and Shuiguang Deng. Exploracoder: Advancing code generation for multiple unseen apis via planning and chained exploration. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 18124–18145, 2025d.

Di Wu, Wasi Uddin Ahmad, Dejiao Zhang, Murali Krishna Ramanathan, and Xiaofei Ma. Repoformer: Selective retrieval for repository-level code completion. *arXiv preprint arXiv:2403.10059*, 2024.

Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. Demystifying llm-based software engineering agents. *Proceedings of the ACM on Software Engineering*, 2(FSE):801–824, 2025.

John Yang, Carlos Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. Swe-agent: Agent-computer interfaces enable automated software engineering. *Advances in Neural Information Processing Systems*, 37:50528–50652, 2024a.

John Yang, Carlos E Jimenez, Alex L Zhang, Kilian Lieret, Joyce Yang, Xindi Wu, Ori Press, Niklas Muennighoff, Gabriel Synnaeve, Karthik R Narasimhan, et al. Swe-bench multimodal: Do ai systems generalize to visual software domains? *arXiv preprint arXiv:2410.03859*, 2024b.

John Yang, Kilian Lieret, Carlos Jimenez, Alexander Wettig, Kabir Khandpur, Yanzhe Zhang, Binyuan Hui, Ofir Press, Ludwig Schmidt, and Diyi Yang. Swe-smith: Scaling data for software engineering agents. *Advances in Neural Information Processing Systems*, 38, 2026a.

John Yang, Kilian Lieret, Jeffrey Ma, Parth Thakkar, Dmitrii Pedchenko, Sten Sootla, Emily McMilin, Pengcheng Yin, Rui Hou, Gabriel Synnaeve, et al. Programbench: Can language models rebuild programs from scratch? *arXiv preprint arXiv:2605.03546*, 2026b.

Daoguang Zan, Zhirong Huang, Wei Liu, Hanwu Chen, Shulin Xin, Linhao Zhang, Qi Liu, Li Aoyan, Lu Chen, Xiaojian Zhong, et al. Multi-swe-bench: A multilingual benchmark for issue resolving. *Advances in Neural Information Processing Systems*, 38, 2026.

Fengji Zhang, Bei Chen, Yue Zhang, Jacky Keung, Jin Liu, Daoguang Zan, Yi Mao, Jian-Guang Lou, and Weizhu Chen. Repocoder: Repository-level code completion through iterative retrieval and generation. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pp. 2471–2484, 2023.

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! *Advances in Neural Information Processing Systems*, 38, 2026a.

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*, pp. 1592–1604, 2024.

Zhe Zhang, Runlin Liu, Aishan Liu, Xingyu Liu, Xiang Gao, and Hailong Sun. Code2bench: Scaling source and rigor for dynamic benchmark construction. In *International Conference on Learning Representations*, volume 2026, pp. 153348–153391, 2026b.

Xinkui Zhao, Rongkai Liu, Yifan Zhang, Chen Zhi, Lufei Zhang, Guanjie Cheng, Yueshen Xu, Shuiguang Deng, and Jianwei Yin. Completion by comprehension: Guiding code generation with multi-granularity understanding. *arXiv preprint arXiv:2512.04538*, 2025.

A Benchmark Construction Details

A.1 Candidate Discovery and Time Scope

The ecosystem screening in Section 2 checks each organization's highest-ranked repositories. We exclude organizations with fewer than two non-archived repositories in this group or no activity within the preceding 24 months.

Cross-repository PR and issue links identify candidate change groups. Test-related paths, such as `tests/`, `spec/`, and `_test`, provide an initial screening signal; behavioral coverage is checked during executable validation. Candidates must be evaluable on Linux. Record-level deduplication uses the ecosystem, source repository, source PR, and involved-repository set.

The review cutoff applies to the latest merge timestamp among the involved PRs, so individual constituent PRs may predate June 1, 2025. The 109,233 cross-referencing PR records count linked activity rather than distinct tasks; link direction does not imply architectural dependency direction.

A.2 Semantic Review, Case Assembly, and Selection

Reviewers examine PR descriptions, linked discussions, changed files, and tests to establish whether the records address one feature or fix and require substantive changes in multiple repositories. An incidental citation does not suffice. Of the 635 candidates retained for environment construction, 240 are runnable and 192 satisfy the per-target F2P requirement.

Record deduplication is distinct from assembling a shared change chain. When multiple candidate records describe the same request, their relevant requirements and repository changes are consolidated rather than released as separate copies. For example, the gRPC timeout interoperability case combines two candidates sharing the same Java PR with two related Go PRs. The construction record regenerates the combined Go reference and test patches and reruns Base/Gold validation. This merge is justified by a common interoperability requirement, not merely by similar titles or overlapping repositories.

The eligible pool contains 60 bug fixes and 132 features. To balance task types while preserving a diverse range of ecosystems and cross-repository changes, we retain all bug fixes and select 60 features, including all 13 three-repository cases and 47 two-repository cases. Feature selection favors substantive changes involving runtime behavior, protocols, schemas, or SDK integration over narrow field forwarding or mechanical updates. Within heavily represented ecosystems, we reduce repeated coverage of similar change types while preserving ecosystem coverage, yielding a balanced release of 120 tasks.

A.3 Dataset Distributions

Coordination patterns. All 120 tasks require coordinated changes across repositories to fulfill a shared feature or bug-fix request. Table 4 summarizes two task-level semantic patterns. *Producer–consumer dependency* accounts for 96 tasks (80.0%): one repository provides capabilities, interfaces, schemas, service behavior, or build artifacts that other repositories consume or adapt to; this category also includes compatibility between protocol endpoints, as illustrated in Figure 14. *Parallel propagation* accounts for 24 tasks (20.0%): multiple repositories implement the same behavior or fix, as in language SDKs, providers, or native and polyfill implementations. Across both patterns, agents must identify the required repository scope and carry the shared request through all necessary implementations. Related implementations can also provide behavioral references and guide adaptations in other repositories, as illustrated by the Sentry SDK case in Appendix C.1.

Table 4: Cross-repository coordination patterns across 120 tasks.

Coordination pattern	Tasks	Share (%)
Producer–consumer dependency	96	80.0
Parallel propagation	24	20.0
Total	120	100.0

Workspace scope. Context repositories are available for reference but are not scored targets. In Figure 6, the horizontal axis gives their number in a task's workspace, and bar heights count tasks with that number. For example, 27 tasks have no additional context repositories, while 93 have at least one. The range is zero to 20, with a median of three per task.

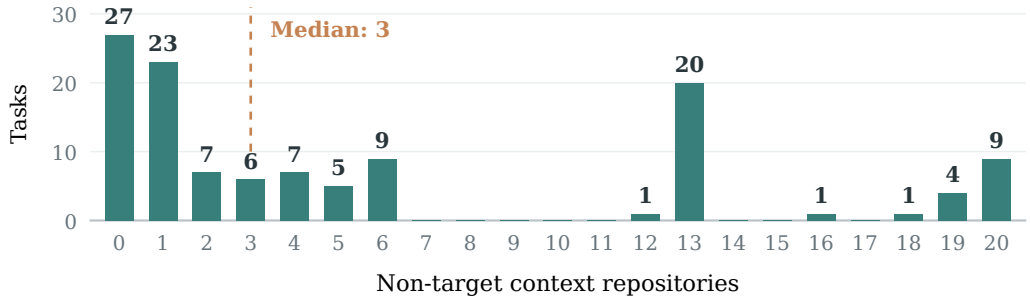
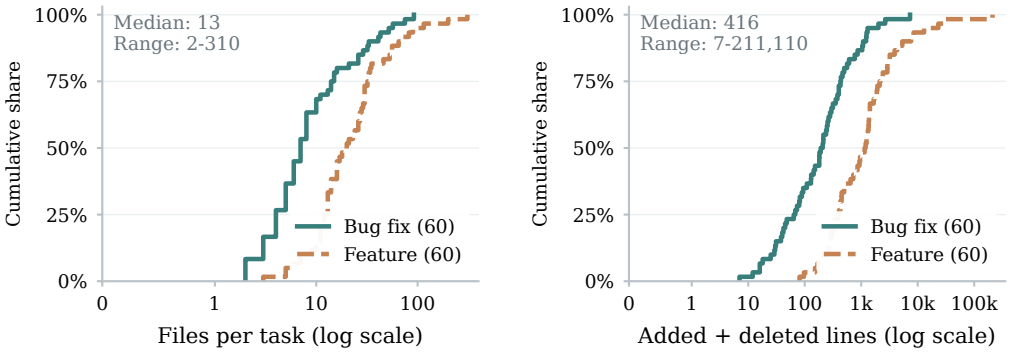


Figure 6: Non-target context repositories available per task across 120 historical workspaces. Bar labels show task counts; the dashed line marks the median. Scored target repositories are excluded.

Reference-patch size. Figure 7 compares the number of files and lines changed by reference patches for bug fixes and features. At each horizontal-axis value, the curve gives the proportion of tasks with no more than that many changed files or lines; a curve farther to the right therefore indicates larger patches. Feature patches are generally larger than bug-fix patches. Across all tasks, the medians are 13 files and 416 lines. Counts include production code, tests, generated files, documentation, and configuration. Line counts sum additions and deletions; binary files contribute only to file counts. Patch size describes the changes, not task difficulty.



(a) Reference patch files **(b) Reference patch lines**

Figure 7: Cumulative reference-patch sizes by task type. Horizontal axes are logarithmic above one and linear near zero.

Test coverage. Figure 8 summarizes 2,815 F2P and 22,139 P2P tests across 253 target-repository instances, using the scoring manifests. The solid curves count tests within each target repository; the dashed curves sum tests across all targets in a task. As above, the vertical axis gives the proportion with no more than the test count on the horizontal axis, not an agent success rate. The per-task curves lie farther to the right because each task combines tests from multiple repositories.

A.4 Prompt Provenance and Language Labels

Prompt construction prioritizes sentences from the original issues and PRs. It removes templates, repeated passages, links, and irrelevant material while retaining observable requirements, reproduction conditions, public interfaces, compatibility constraints, and necessary edge cases. Case records map

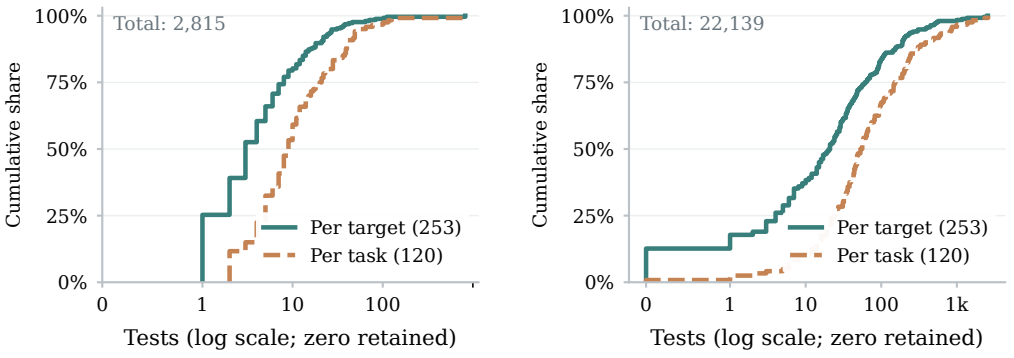


Figure 8: Cumulative test counts per target repository and per task. Horizontal axes retain zero counts and use the same scale as Figure 7.

each prompt paragraph to its source and identify any consolidation or necessary clarification. The Gold implementation is not expanded into a step-by-step solution; private symbols, file layouts, and algorithms are not added merely because that implementation uses them. Test-required behavior must be supported by the public demand and expressed sufficiently for a correct implementation, rather than hidden behind an undisclosed naming or representation choice.

Language-family labels use reference production-code changes, excluding tests, documentation, configuration, conventional build-support files, and generated bundles. A task spanning multiple families may do so within a repository or across repositories. The resulting 86/34 split describes language diversity in the required work, not a controlled intervention on language.

A.5 Prompt Construction Example

One author constructed the prompts, and two other authors independently reviewed each prompt against its source issues and pull requests to verify that all required behavior was preserved and no implementation-specific instructions were introduced. Disagreements were resolved through discussion.

Figure 9 compares the source passages with the complete prompt for the Expo logout task. eas-cli#3555 supplies the problem description and behavioral requirements. expo#45802 explicitly requests parity with that change and repeats the failure-handling requirement, establishing that the request also applies to Expo CLI. The shared prompt consolidates these overlapping requirements rather than repeating the same behavior once per repository.

The first paragraph is unchanged from the PR. The second removes a PR link and capitalizes "CLI." The third is reworded to preserve the shared logout and failure-handling requirements while omitting the specific command and endpoint, leaving agents to determine how to implement the behavior from each repository's code.

A.6 Historical Workspace Snapshots

Each target repository uses the base commit of its corresponding PR. Context repositories use snapshots no later than the earliest target base to prevent future-information leakage. Gold patches and hidden tests are not exposed to agents.

A.7 Evaluator Test Isolation and Patch Conflicts

Agent changes are evaluated on clean Base snapshots, with evaluator-owned tests supplied separately. When a hidden-test patch targets a public test file that the agent has also edited, the evaluator can materialize the hidden version from the clean Base in a temporary directory and copy only the patch's target files into the evaluation workspace. Test-patch assembly therefore does not depend on whether

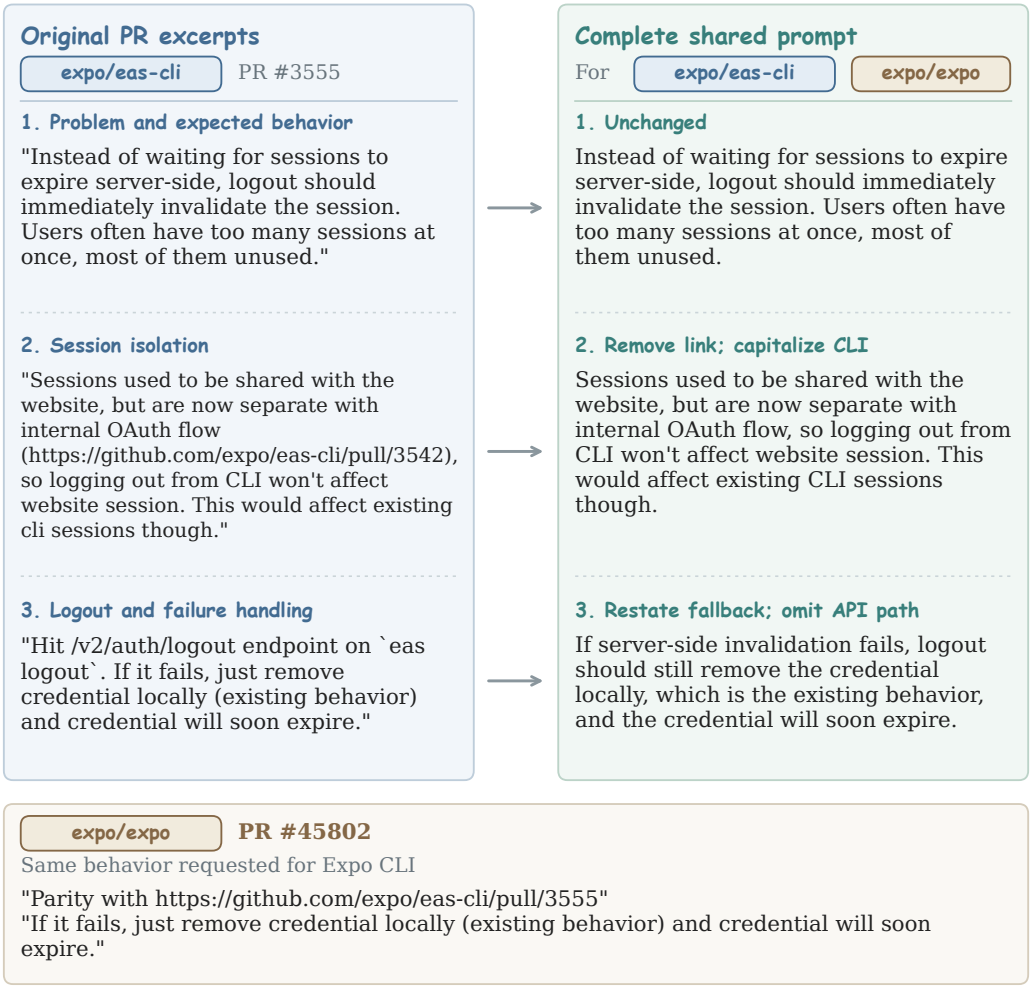


Figure 9: Prompt construction for the Expo logout task. Left: wording from eas-cli#3555. Right: the complete shared prompt. Bottom: expo#45802 establishes the same requirement in the second repository.

the agent's public-test edits happen to match the patch context. Files outside the evaluator patch targets are not changed by this overlay.

For profiles that relocate upstream tests into separate hidden files, the evaluator materializes the source test from Base and installs it at the configured hidden path. Where compilation would otherwise encounter duplicate test declarations, the enclosing test type is renamed while assertions and test bodies are preserved. Overlapping evaluator patches are applied in their configured order so that a later patch does not discard an earlier evaluator layer. Patch-injection failures are recorded and cannot produce a successful evaluation. These mechanisms address test-file and declaration conflicts, not behavioral acceptance criteria or an agent's production-code implementation.

Section A.8 documents revisions to test acceptance criteria separately from these test-installation mechanisms.

A.8 Examples of Requirement-Aligned Test Review

A.8.1 Relax Implementation-Specific Constraints

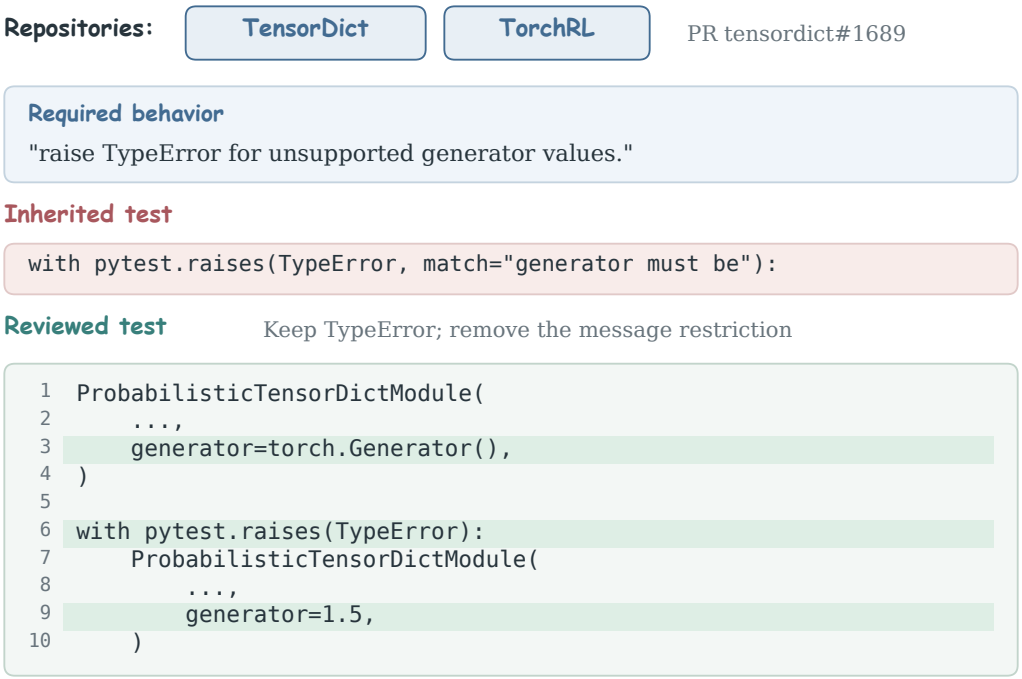


Figure 10: Preserve the required exception without fixing its wording. Excerpts normalize formatting and quotation marks; line numbers are local and ellipses mark omissions.

Request and scope. The TensorDict/TorchRL task adds a local random generator for probabilistic sampling and forwards the option through the actor interface. The TensorDict PR explicitly lists `invalid type raises TypeError` among its requirements (tensordict#1689). The final prompt likewise states:

> Preserve the existing behavior when `generator=None`, and raise `TypeError` for unsupported generator values.

Neither statement specifies the error message.

Inherited test. The invalid-input test passes a floating-point value and checks both the exception type and the phrase `generator must be` (Figure 10).

Audited test. The test first confirms that a supported generator is accepted, then checks that the same call with an unsupported value raises the required exception. Removing the message match permits different explanations of the same input error.

A.8.2 Remove Unrequested Functionality Checks

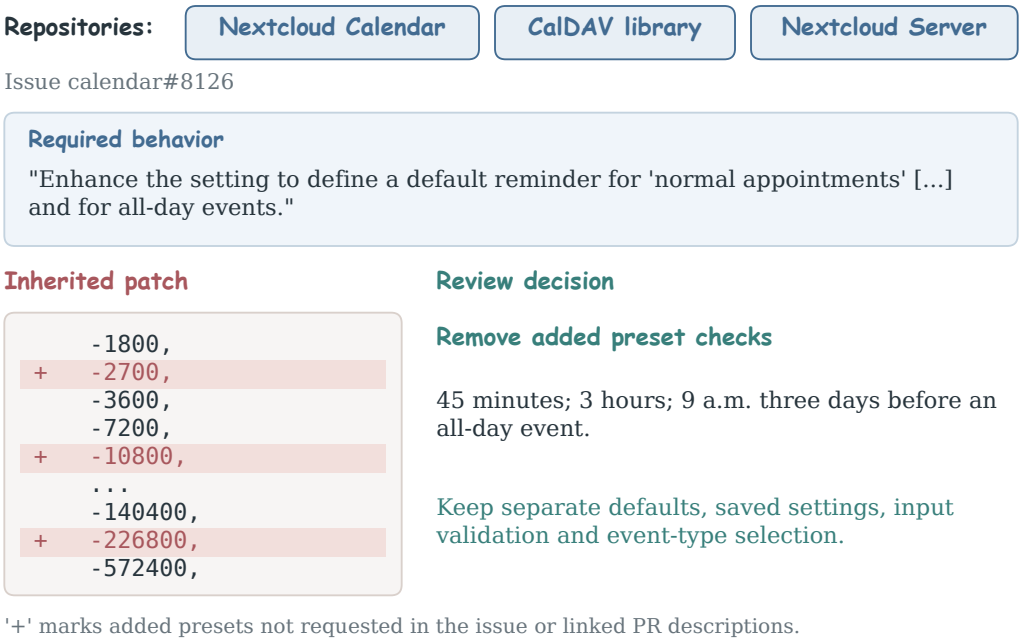


Figure 11: Remove unrequested preset expansion. Excerpts normalize formatting and quotation marks; ellipses mark omissions.

Request and scope. The Nextcloud task spans Calendar, its CalDAV client library, and Server. Previously, one default reminder applied to both timed and all-day events. The issue requests separate settings:

> Enhance the setting to define a default reminder for ”normal appointments” (not sure what would be a good name here) and for all-day events.

This passage is from calendar#8126, linked by calendar#8144, cdav-library#1012, and server#59517. The prompt specifies separate settings, their storage and propagation, input validation, and selection according to event type. It does not require additional built-in reminder presets.

Inherited test. Alongside the requested settings, the reference patch expands the preset lists. The inherited test patch adds `-2700`, `-10800`, and `-226800` to the expected arrays in `defaultAlarmProvider.test.js` (Figure 11). The first two additions introduce 45-minute and three-hour presets. The third introduces a reminder at 9:00 a.m. three days before an all-day event. These additions are present in the reference implementation, but are not requested in the issue or linked PR descriptions.

Audited scope. We remove the preset-expansion hunks from the hidden patch rather than require these three additional choices. This does not remove support for user-configured offsets or the requirement to keep separate timed-event and all-day defaults. Tests for the DAV properties, client-model fields, saved settings, input validation, and event-type-specific behavior remain. The preset test file remains in the test profile; existing preset coverage is retained without making the added choices a task requirement.

Validation. Both revised cases retain failing target behavior on Base and pass with Gold. The TensorDict/TorchRL case has 21 F2P and 580 P2P checks; the Nextcloud case has 53 F2P and 223 P2P checks.

Each test revision was first proposed by one author and then independently assessed by two other authors with reference to the task requirements and the original issues/PRs. Revisions were accepted only when both reviewers judged that they removed implementation-specific or unrequested constraints without weakening the required behavior. Any disagreement was settled through discussion.

Revision coverage. Across the 120 tasks, requirement-aligned test revisions affected 88 tasks. Implementation-specific constraints were relaxed in 87 tasks, and unrequested functionality checks were removed in 16 tasks; 15 tasks involved both categories. These are task-level counts, not counts of individual tests or assertions, and exclude changes limited to test installation, fixtures, or execution environments.

B Supplementary Evaluation Results

B.1 Joint and Independent Repository Comparison

For the 89 prompt-equivalent cases in Section 3, **Joint** exposes the complete ecosystem workspace to one run, while **Independent** exposes only one target repository to each repository-specific run. Both use the same agent configuration, tool permissions, and per-run limits, but Independent receives one full run per target repository. This follows the conventional single-repository benchmark setup, in which each repository-level task receives its own run with the full per-run budget. The comparison therefore matches per-run limits, not the total budget available for each multi-repository task. Our aim is to examine whether solving repositories separately alleviates cross-repository difficulties, not to isolate the effect of execution scope under equal total budgets.

We pair outcomes for the same repository within the same case. Repository success requires all F2P and P2P tests to pass. At case level, Joint succeeds when one run resolves every target repository; Independent-all succeeds when every repository-specific run succeeds. Table 5 reports the paired outcome percentages underlying Section 4.3.

Table 5: Paired Joint–Independent outcomes.

Outcome	Cases (%)	Repositories (%)
Both succeed	26.97	51.83
Joint only	13.48	9.95
Independent only	8.99	8.90
Neither succeeds	50.56	29.32

Recovery among Joint-failed repositories. Table 6 groups the 73 repositories that fail in Joint by whether the final Joint diff records a modification. This grouping does not use the Independent outcome, and a recorded modification does not imply a correct implementation. Recovery requires all F2P and P2P tests to pass in Independent. Because the groups differ in task composition, their recovery-rate difference is descriptive, not a causal effect of assigning a separate run.

Table 6: Independent recovery among Joint-failed repositories.

Joint modification	n	Independent pass	Recovery (%)
None	20	12	60.00
Present	53	5	9.43

Reversals by task intent. Among the 29 bug fixes, nine cases succeed in both settings, one only in Joint, four only in Independent-all, and 15 in neither. Among the 60 features, the corresponding

counts are 15, 11, four, and 30. The opposite net directions should not be generalized beyond this sample.

B.2 Test-Level Pass Rates

Tables 7 and 8 report the proportion of individual tests passed, computed as total passed tests divided by total tests in each group. Unlike the repository-level metrics in the main text, these rates capture partial progress within repositories and weight repositories by their test counts.

Configuration abbreviations in Table 7 follow Table 1. Table 8 uses the same 2,532 F2P and 18,051 P2P tests in each condition for the 89 paired RQ3 tasks.

Table 7: RQ1 test-level pass rates (%).

	Bugfix		Feature		Overall	
Configuration	F2P	P2P	F2P	P2P	F2P	P2P
GPT + Codex	75.00	99.89	83.57	98.01	81.74	98.84
GPT + CC	62.83	99.69	80.05	87.44	76.38	92.85
Opus + CC	74.33	98.80	39.82	87.96	47.18	92.75
Gemini + CC	31.00	99.77	2.62	95.13	8.67	97.18
DeepSeek + CC	70.67	98.55	71.56	94.71	71.37	96.40
Qwen + CC	73.67	98.78	75.03	88.69	74.74	93.15
GLM 5.3 + CC	52.83	93.77	63.34	92.74	61.10	93.19

Table 8: RQ3 test-level pass rates (%).

Condition	F2P	P2P
Joint	81.52	98.63
Independent-all	79.90	97.86

B.3 Failure Annotation Protocol

We classify each unresolved run using a two-stage procedure. First, we inspect the final diffs of all target repositories. If all target repositories receive substantive code changes but the run still fails the required behavior or regression checks, the run is classified as Post-edit failure. Otherwise, we inspect the execution trajectory to distinguish the remaining two categories. If the agent fails to identify any required repository or any required change necessary to complete the task, we classify the run as Incomplete scope identification. If the agent explicitly recognizes the necessary work but does not deliver the corresponding change, we classify it as Recognized work without delivery.

Two authors independently annotated the unresolved runs according to these rules, achieving 91.2% agreement (Cohen's $\kappa = 0.86$). Disagreements were resolved through discussion.

C Trajectory Evidence for Cross-Repository Mechanisms

C.1 Cross-Repository Behavior in Joint Runs

Sentry SDKs. The task requires implementing the same strict trace-continuation policy in the Go, Python, and Ruby SDKs. Codex declares, "The target is the Go SDK," and later reads Python tracing code as a reference. Only Go receives a patch, passing 14/14 F2P checks; Python and Ruby pass 0/22 and 0/13. Although the SDKs have separate implementations, completing the shared request requires the agent to identify and update all three target repositories. Work on one repository can also draw on another repository's implementation, as the use of Python code while working on Go illustrates.

Laravel AI/MCP. In Laravel AI/MCP, the agent instead moves the streaming fix into the Framework context repository. AI passes 2/2 F2P checks, but MCP remains unchanged at 0/3. Editing a context

repository is allowed; the unresolved target behavior, rather than that location choice itself, makes the task incomplete.

Swagger. In Swagger, Codex concludes that the library's serializer "already repeats actual JavaScript arrays correctly." It fixes only the UI, which passes 2/2 checks; the library remains at 0/5 because its public request-building path still mishandles the inputs.

Sentry CLI/Dart. Codex identifies the CLI parser's missing global URL option but does not identify the Dart plugin's required adaptation. It ends with an environment-variable workaround and a suggested CLI fix, leaving both repositories unmodified. Diagnosing one repository's defect does not establish the full modification scope.

Godot/Native. In Godot/Native, Codex explicitly states that the fault is in the Godot call site, then implements only the Native API (Figure 12). Native passes 11/11 F2P checks while Godot remains at 0/1.

Kubernetes. Figure 13 shows an incomplete Kubernetes consumer migration despite edits to both repositories. Preserving the generated schema names did not ensure that the downstream code used them correctly.

Incomplete propagation within edited repositories. In the Sentry Python/Ruby task, the agent edits Python's synchronous and asynchronous HTTPX integrations and Ruby's shared HTTP helper. Its targeted HTTPX command passes four tests, and the Ruby check also passes. However, Python's Celery integration still overwrites existing baggage: the evaluated message loses `custom=value`. Python passes 2/3 F2P checks and Ruby 1/1. The shared requirement was propagated across repositories but not across all relevant integrations within them.

A shared representation can still violate the contract. In the Sentry/Symbolicator Java-symbolication task, the agent changes both the request producer and consumer to use an exception-name string. The required stacktrace interface instead carries a structured exception with a type and optional module. The agent's local Symbolicator suite passes 18 tests, but its integration helper leaves the new stacktrace exception at its default value, while the added remapping test supplies a string directly. Neither validates a structured exception entering the new field. Formal requests then fail with `invalid type: map, expected a string`; Symbolicator passes only 1/6 F2P checks, although the other two target repositories pass theirs. This is not disagreement between the edited endpoints: both encode the same mistaken interpretation, and their local checks fail to expose it.

Connecting an interface without preserving its contract. The Sentry wizard installs the SDK's new error handler on React Router, but the handler treats a wrapped callback argument as direct React error information, losing the component stack. In Symfony's hydration migration, the agent delegates instantiation to the new DeepClone API but exposes its exceptions directly, replacing the public Symfony exceptions expected by existing callers. In both cases, the new call is present, yet the adapter fails to preserve the required input or error behavior.

Sentry: completing two repositories while breaking the third adapter. The task adds `log_flush_threshold` to the PHP SDK and its Laravel and Symfony integrations; the option must accept `null` or a positive integer. Claude Code with DeepSeek V4 Pro identifies and modifies all three repositories (Figure 15). The SDK accepts `null`, but the Symfony configuration uses an integer-only node. The run reports completion, yet formal checks reject the required `null` setting in Symfony. PHP and Laravel pass 13/13 and 4/4 F2P checks, respectively, while Symfony passes 22/24. This failure concerns the correctness of the remaining adaptation, not an overlooked repository.

Findings and implications. The cases distinguish identifying necessary changes, covering relevant execution paths, and preserving the contracts between repositories. Godot/Native leaves the diagnosed caller unchanged; the Python/Ruby task leaves an integration unfinished inside an edited repository; and Sentry/Symbolicator adopts a consistent but incorrect request representation. Conversely, gRPC

succeeds by applying complementary, rather than identical, rules at the two endpoints (Figure 14). Neither repository coverage nor agreement between edited components alone establishes coordinated completion.

C.2 Joint versus Independent Codex Runs

These three tasks illustrate omitted-work recovery and two uses of cross-repository context under the paired setup in Appendix B.1.

Ansible: using the other repository to guide implementation. The task requires metrics-utility to collect execution information from metrics-service. The Joint run reads the service's task-execution model before implementing the utility query. Its query uses the service's actual task and execution tables and the `started_at` field. The independent utility run searches for the service model but submits a query using a different table and a `started` field instead. Figure 17 contrasts the inspected model with the submitted queries. This illustrates how another repository supplies concrete implementation facts, rather than only additional work to complete.

Findings and implications. Separate assignments recover omitted SDK work in Sentry. In Ansible, the related repository supplies facts needed to implement a compatible query; in Symfony, it provides a behavioral reference that helps reveal an implementation error. The cases distinguish completing overlooked work from using cross-repository information to implement and check that work correctly.

C.3 Codex CLI versus Claude Code with GPT-5.6-sol

The Rails/Propshaft pair holds the model and prompt fixed. It illustrates a difference in scope checking between two runs, rather than isolating a particular framework component.

C.4 Model Comparisons within Claude Code

Each comparison holds Claude Code and the task prompt fixed while varying the LLM. Fleet illustrates the Delivery failure category in RQ1: recognized work is not implemented. Rust/Cargo, PyPA, and TensorDict/TorchRL illustrate Post-edit failures: the target repositories are modified, but the changes do not jointly satisfy the request. These selected runs illustrate the failure mechanisms rather than establish model-wide rankings.

Fleet: carrying a plan through to implementation. The task requires Elastic Agent to compress checkin requests and Fleet Server to accept them. Gemini plans both changes but then makes 113 read or search calls without implementing either side. Opus implements client compression and server decoding, then checks that compressed and uncompressed requests behave equivalently (Figure 20). Gemini passes 0/5 and 0/2 F2P checks in the two repositories; Opus passes 5/5 and 2/2. The contrast is between recognizing the required work and carrying it out, not between identifying different target repositories.

Rust/Cargo: covering the consumer's entry points. The task requires Rust bootstrap to adopt Cargo-managed warning denial. Opus and GPT both modify Cargo and Rust, but Opus treats `bootstrap.py` as a separate path and leaves it unchanged, whereas GPT includes it in the migration (Figure 21). Opus passes Cargo's evaluation but fails Rust's F2P check; GPT passes both repositories. DeepSeek also migrates this entry path and passes both. Because the failing Opus run edits both target repositories, this is incomplete implementation within an edited repository, not failure to identify a target repository.

PyPA: checking exchanged metadata. The task distinguishes an absent `Import-Name` field from an explicitly empty one. Opus feeds the producer's output into the parser and checks both cases. DeepSeek modifies both repositories and reports passing local suites, but its emitter drops the empty field (Figure 22). Opus passes both repositories; DeepSeek leaves both F2P-incomplete.

TensorDict/TorchRL: checking downstream state updates. The task requires sampling to write a fresh seed back to the specified TensorDict key. Qwen probes seed updates in TensorDict and

Godot / Native

Codex CLI
GPT-5.6-sol / Joint

Repositories: sentry-godot | sentry-native

Request

Log messages containing printf format specifiers crash on Windows.
...
The native logging contract must provide `sentry_log(level, body, attributes)` for plain-string messages.

1 Locate the Godot call site and identify the fault

item_3 Command (Godot)
```
sed -n '180,270p' src/sentry/native/native_sdk.cpp
```

item_3 Returned code (Godot)
```
sentry_log_info(p_body.utf8(), attributes);
```

item_4 Agent diagnosis
The fault is in the Godot call site, but the requested API contract belongs in the `sentry-native` submodule.

2 Add the plain-string API, but leave its caller unchanged

Patch (Native SDK)
```
+SENTRY_EXPERIMENTAL_API log_return_value_t sentry_log(
+    sentry_level_t level, const char *body, sentry_value_t attributes);
```

Diff summary
```
sentry-godot: 0 changed files
sentry-native: 4 changed files
```

3 Finish after the Native SDK unit suite passes

item_30 Command (Native SDK)
```
ctest --test-dir build --output-on-failure
```

item_30 Local test output (Native SDK)
```
100% tests passed, 0 tests failed out of 304
```

item_33 Final response
Implemented in `sentry-native`.

Formal evaluation	F2P	P2P
sentry-godot	0/1	14/14
sentry-native	11/11	0/0

The agent identifies the faulty Godot caller but leaves it unchanged. A working SDK API does not repair a caller that still uses the old logging path.

Figure 12: Godot/Native. The agent identifies the faulty Godot call site but implements and tests only the Native SDK API, leaving the caller unchanged.

Kubernetes

Codex CLI
GPT-5.6-sol / Joint

Repositories: kube-openapi | kubernetes

Request

Migrate the Kubernetes API to use this generator while keeping all OpenAPI model names the same.

1 Migrate both the generator and its consumer

item_35 Agent response

Next I'm applying the same option to Kubernetes' vendored generator and codegen script, then generating the API accessor files.

2 Check the generated schema-name set

item_84 Command

```
GOPROXY=off go run -mod=vendor ./pkg/generated/openapi/cmd/models-schema
```

item_84 Comparison command

```
actual=set(d); print(f\"expected={len(expected)} actual={len(actual)}
missing={len(expected-actual)} extra={len(actual-expected)}\")
```

item_84 Tool output

```
expected=1362 actual=1362 missing=0 extra=0
missing names: []
extra names: []
```

3 Report successful migration and focused tests

item_115 Final response

- Migrated Kubernetes codegen, producing 90 model-name files.
- Preserved all 1,362 OpenAPI model names exactly.
- Updated definition naming to avoid re-transforming declarative names.
- Integration and focused Kubernetes tests pass.

Formal evaluation	F2P	P2P
kube-openapi	1/1	0/0
kubernetes	776/824	1367/1370

Equal name sets verify one invariant, not every downstream type-to-name mapping. The formal evaluation still finds incomplete consumer behavior.

Figure 13: Kubernetes. Both generator and consumer are modified, but preserving the schema-name set does not establish complete downstream migration.

gRPC / Go + Java

Codex CLI
GPT-5.6-sol / Joint

Repositories: grpc-go | grpc-java

Request

Handle gRPC timeout header interoperability across Go and Java.

1 Distinguish outgoing values from legacy input

item_5 Agent response

The current Go decoder already accepts zero, but it also accidentally accepts negative values; Java emits `0n` for zero.

2 Apply complementary, not identical, policies

item_13 Agent response

Java now floors emitted timeouts at one nanosecond, while Go validates the numeric field as digits-only but deliberately accepts zero.

3 Test both implementations

item_14 Command (Go)

```
go test ./internal/transport
```

item_14 Local test output (Go)

```
google.golang.org/grpc/internal/transport   11.552s
```

item_17 Command (Java)

```
./gradlew -PskipAndroid=true :grpc-core:test --tests
io.grpc.internal.GrpcUtilTest
```

item_17 Tool output (Java)

```
BUILD SUCCESSFUL in 22s
```

4 Report both sides of the compatibility contract

item_20 Final response

- Java emits `1n` instead of `0n` for zero/expired deadlines.
- Go accepts legacy `0S` while rejecting signs, malformed values, and over-eight-digit values.

Formal evaluation

	F2P	P2P
grpc-go	1/1	109/109
grpc-java	1/1	94/94

Coordinated changes need not enforce identical behavior: new Java output is positive, while Go retains compatibility with legacy zero-valued input.

Figure 14: gRPC. Compatibility requires complementary policies: Java emits positive timeouts, while Go accepts legacy zero-valued input. Both repositories pass formal evaluation.

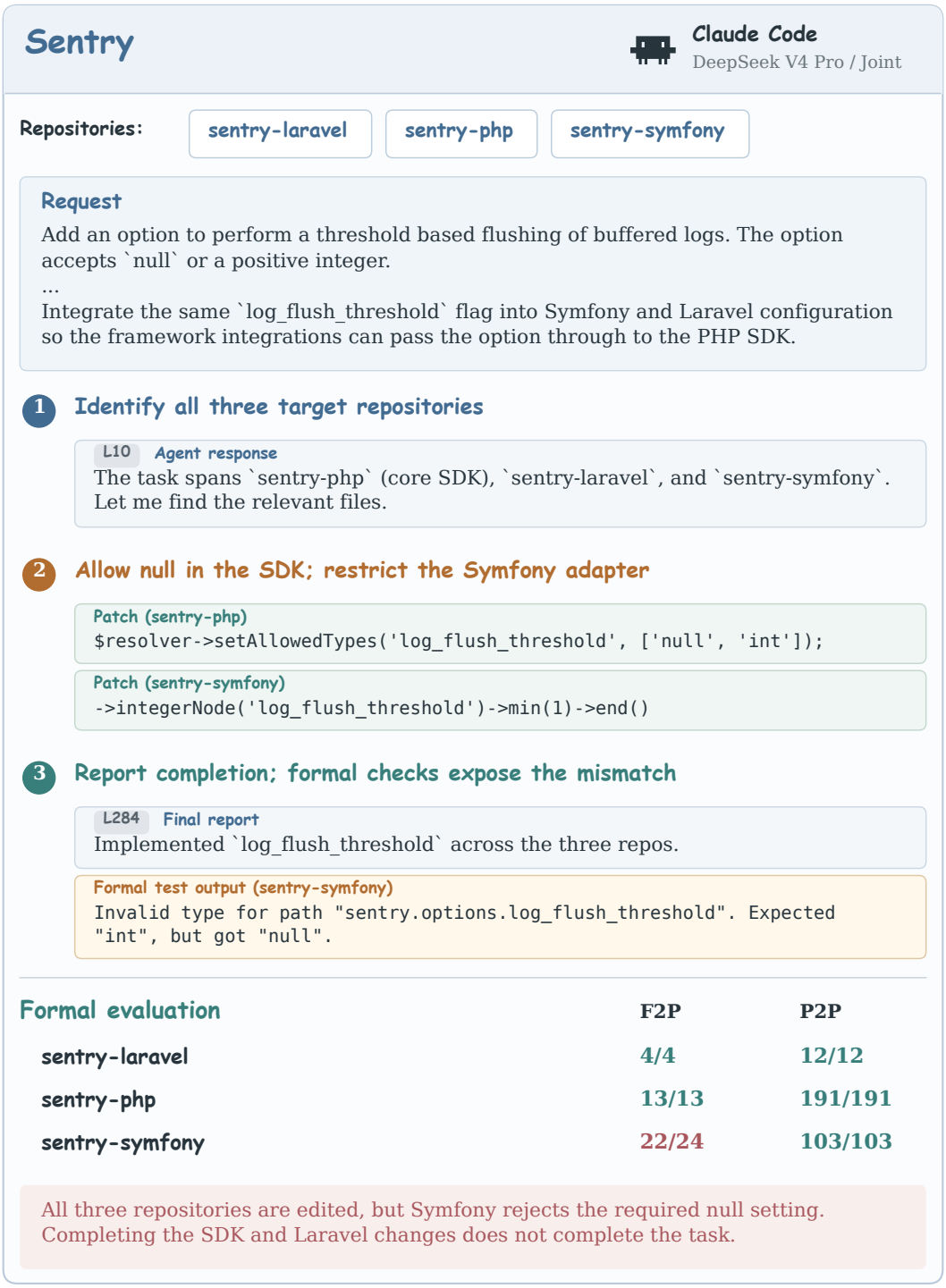


Figure 15: Sentry configuration with Claude Code–DeepSeek V4 Pro. All three repositories are edited, but Symfony rejects the required `null` setting while the PHP SDK and Laravel pass.

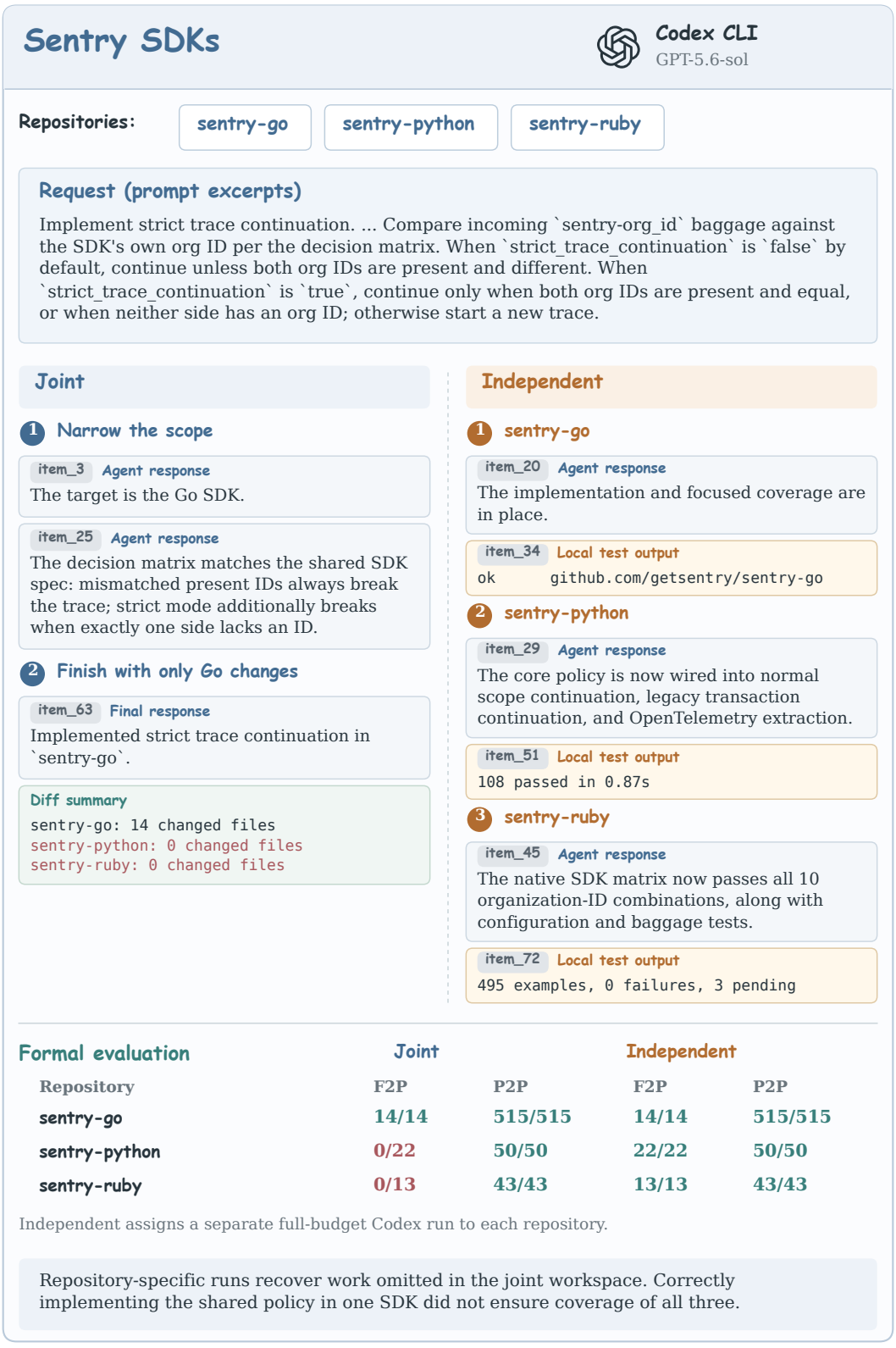


Figure 16: Sentry. The joint run implements a shared policy only in Go. Independent runs complete all three SDKs, recovering the previously omitted Python and Ruby work.

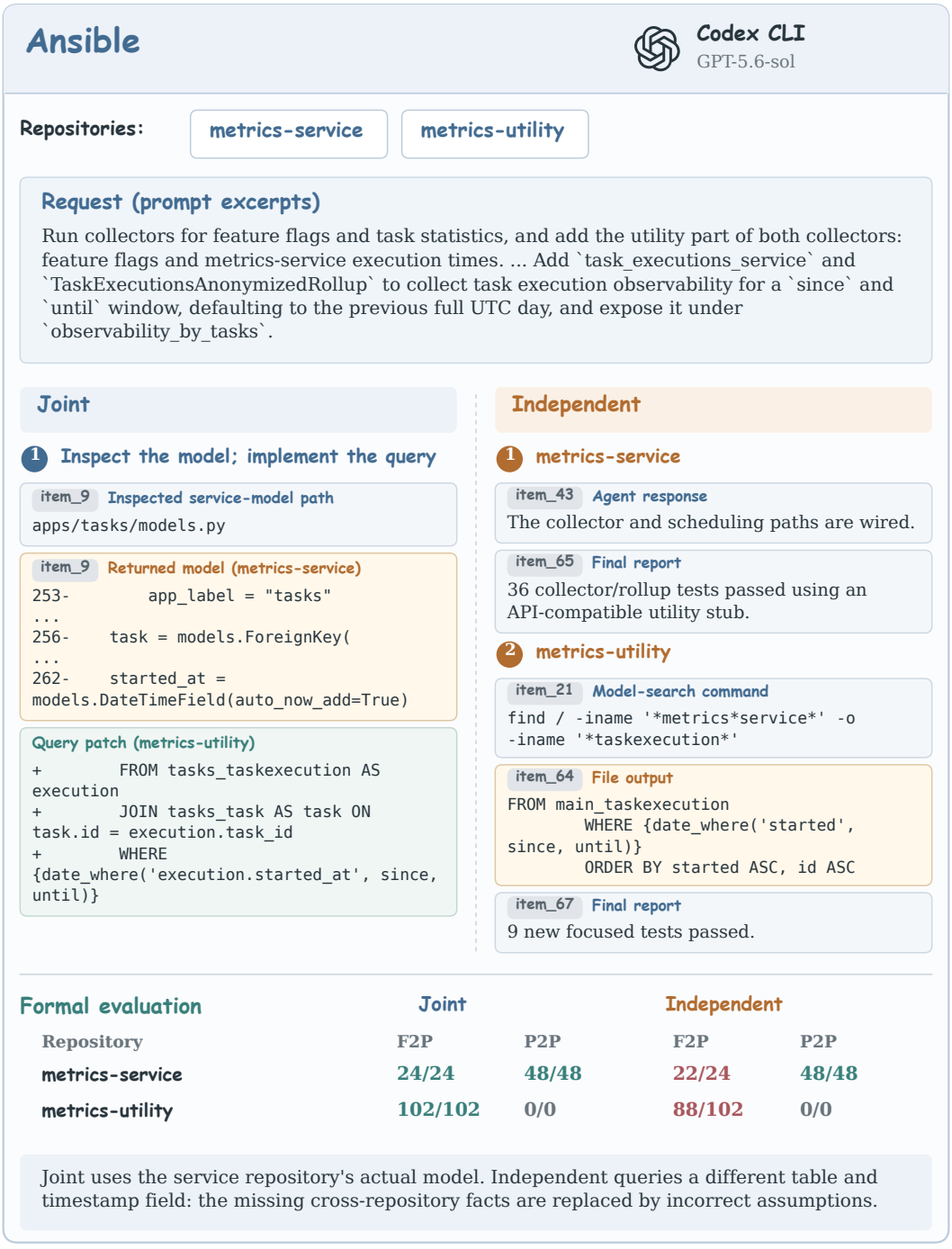


Figure 17: Ansible. The Joint run reads the service repository's model and implements a compatible utility query. The independent utility run submits a query inconsistent with that model.

Symfony DeepClone

Codex CLI
GPT-5.6-sol

Repositories: php-ext-deepclone | polyfill

Request (prompt excerpts)

Keep Symfony DeepClone v0.3.0 hydrate behavior equivalent between the native extension and the PHP polyfill. ... In `deepclone_from_array()`, reject a property scope that is not a loaded class name. ... Align typed-property hydration with raw-value engine semantics so wrong-type writes into typed slots are no longer silently allowed. Preserve private-shadowing roundtrip semantics through `to_array` / `from_array`.

Joint

1 Compare implementations

item_22 Agent response
The native extension already has loaded-scope validation in `from_array`; the PHP path does not, so that will be tightened.

2 Test native and PHP paths

item_138 Command (native)
```
php -d extension=.../deepclone.so...
tests/DeepClone/DeepCloneTest.php
```

item_139 Command (polyfill)
```
php ... tests/DeepClone/DeepCloneTest.php
```

items 138 + 139 Local test output
native + polyfill
```
Tests: 158, Assertions: 294, Skipped: 2.
```

3 Correct typed-reference coercion

item_150 Agent response
I found and corrected one subtle mismatch in typed reference coercion:

Independent

1 php-ext-deepclone

item_6 Agent response
The checkout is the extension repository only, with no separate Symfony component tree or existing PHP polyfill.

item_74 Agent response
I also verified typed references remain references while acquiring engine type-source enforcement.

item_102 Final report
25 tests passed, 2 skipped

2 polyfill

item_5 Agent response
The repository currently has the pre-v0.3 three-argument hydrate API, so the core change is a compatibility migration rather than a small patch.

item_23 Agent response
I'm wiring hydration through that API while keeping ordinary userland writes reference-preserving.

item_57 Local test output
```
Tests: 166, Assertions: 310, Skipped: 2.
```

Formal evaluation

	Joint		Independent	
Repository	F2P	P2P	F2P	P2P
php-ext-deepclone	9/9	0/0	6/9	0/0
polyfill	31/31	150/150	29/31	150/150

Independent assigns a separate full-budget run per repository; this contrast is not a causal estimate.

The successful joint run compares related implementations and checks both. Independent runs make local progress but do not fully satisfy either repository's requirements.

Figure 18: Symfony. The successful joint run compares native and PHP behavior and tests both implementations. Independent runs pass local checks but leave required behavior incomplete in both repositories.



Figure 19: Rails/Propshaft. Codex revisits the request after Propshaft tests pass and extends the change to Rails' static-file middleware. Claude Code with the same GPT model modifies only Propshaft.

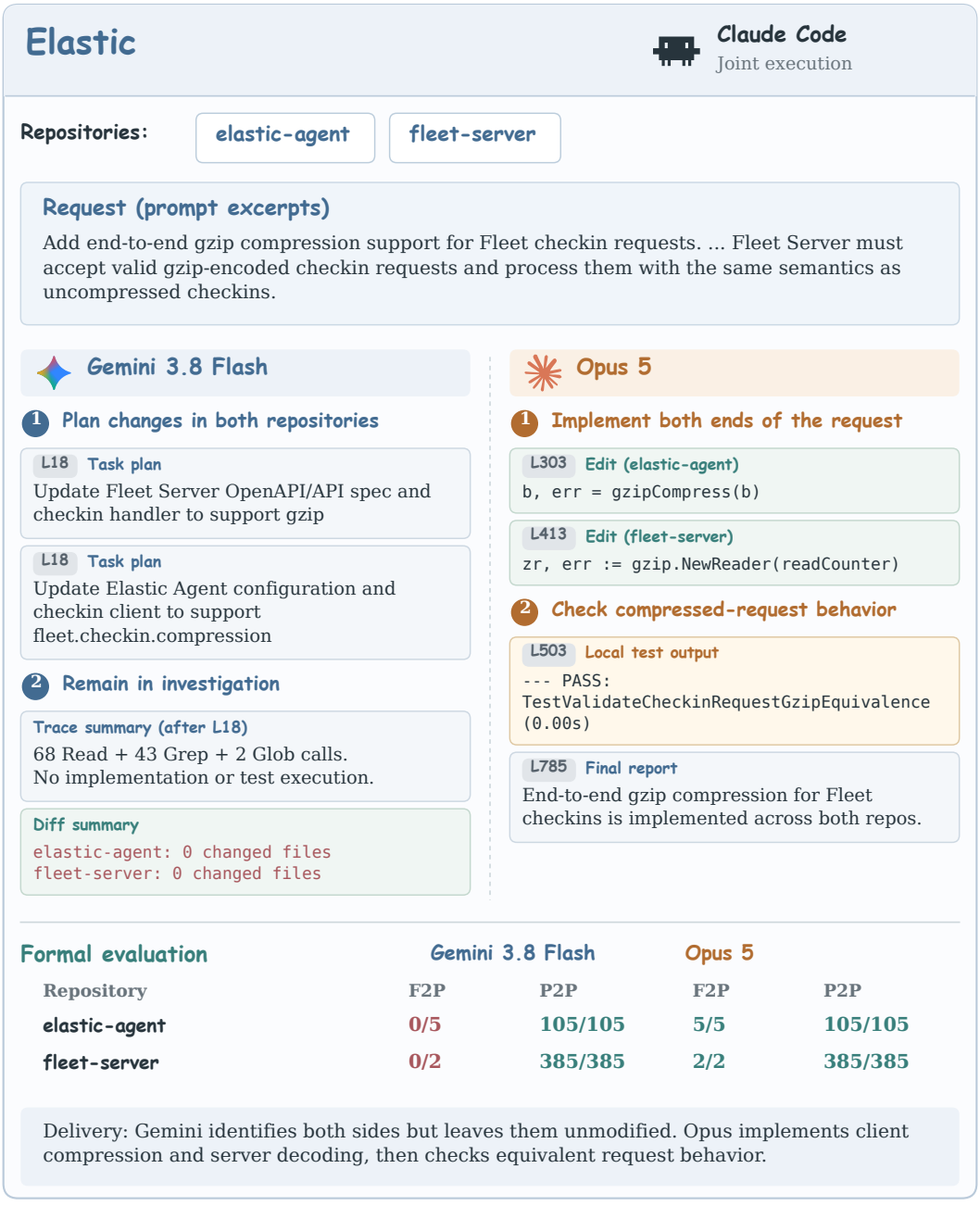


Figure 20: Elastic with Claude Code. Gemini plans changes in both repositories but remains in investigation. Opus implements client compression and server decoding, checks equivalent request behavior, and passes both repositories.



Figure 21: Rust with Claude Code. Both runs complete the Cargo change. Opus leaves the Python bootstrap entry unchanged, while GPT includes it in the migration and passes both repositories.

PyPA

Claude Code
Joint execution

Repositories: packaging | pyproject-metadata

Request (prompt excerpts)

Implement PEP 794 Import Name Metadata support for METADATA 2.5. ... Preserve the PEP distinction between no entry and an empty `Import-Name` entry; an explicitly empty entry still requires metadata version `2.5`.

Opus 5

1 Pass emitted metadata to the parser

L71 Command

```
text = str(md.as_rfc822())
    parsed = Metadata.from_email(text,
validate=True)
```

2 Check empty and absent separately

L72 Local test output

```
explicitly empty import-names:
  emitted version = 2.5
  parsed  version = 2.5
  import_names      = []
```

L72 Local test output

```
absent:
  emitted version = 2.1
  parsed  version = 2.1
  import_names      = None
```

DeepSeek V4 Pro

1 Iterate names; emit nothing if empty

Patch (pyproject-metadata)

```
if self.import_names is not None:
    for import_name in self.import_names:
        smart_message["Import-Name"] =
import_name
```

2 Local suites miss the empty field

L263 Final report

Test results: pyproject-metadata 216 passed; packaging 27295 passed.

Formal test output (pyproject-metadata)

```
Right contains one more item:
('Import-Name', '')
```

Formal evaluation

	Opus 5		DeepSeek V4 Pro	
Repository	**F2P**	**P2P**	**F2P**	**P2P**
packaging	27/27	263/263	23/27	263/263
pyproject-metadata	22/22	183/183	21/22	183/183

The excerpts isolate empty-field behavior; DeepSeek also fails two private-marker whitespace checks.

Post-edit: changing both repositories does not preserve the meaning of exchanged metadata. The empty-list emitter loses the distinction between empty and absent.

Figure 22: PyPA. Opus checks that metadata emitted by pyproject-metadata retains its meaning when parsed by packaging. DeepSeek changes both repositories and reports passing local suites, but its emitter drops explicitly empty import-name fields.

through a TorchRL actor. DeepSeek also edits both repositories, but its integer-seed write-back raises an error in both layers (Figure 23). Qwen passes both repositories, while DeepSeek leaves both F2P-incomplete.

D BENCHMARK SCOPE AND LIMITATIONS

WIDESWE tasks are mined from real-world cross-repository changes and involve two or three target repositories. They cover diverse coordination patterns, including shared feature implementation across language SDKs, adaptations between libraries and downstream integrations, and interface changes between producers and consumers. Tasks are evaluated in Linux environments. The results do not establish how agents perform on tasks involving more repositories or requiring other platforms.

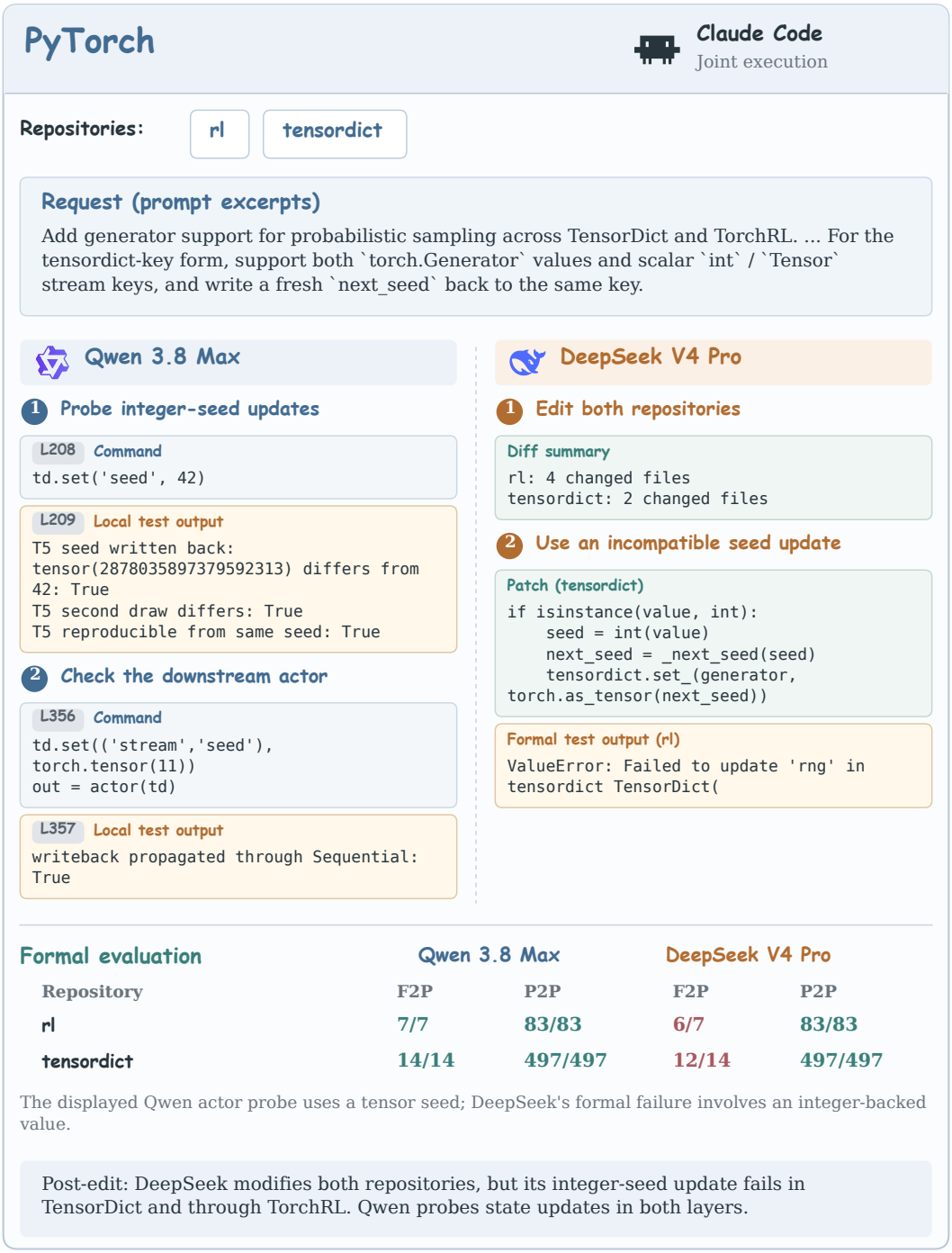


Figure 23: TensorDict/TorchRL. Qwen probes seed updates in the library and through its downstream actor. DeepSeek also modifies both repositories, but an integer-seed write-back error affects both layers, illustrating how a shared-state defect propagates across repository boundaries.