CODEV-BENCH: CAN CODING AGENTS HANDLE NATURALLY CONCURRENT DEVELOPMENT?

Anonymous authors
Paper under double-blind review

ABSTRACT

Current coding benchmarks usually assume a single developer working on a standalone issue. However, real-world software development is inherently concurrent and collaborative, with multiple developers authoring pull requests (PRs) within the same codebase. To bridge this gap, we introduce CODEV-BENCH[1], a novel benchmark designed to evaluate coding agents on concurrent multi-task resolution. CODEV-BENCH involves 206 concurrent PR-pair tasks drawn from 137 production-scale open-source repositories across six languages. Each task consists of two independently authored PRs that branch from a shared base commit and are subsequently integrated into the main branch. The benchmark reflects the complexity of real concurrent development, with an average of 17.1 non-test files and 507 lines of code modified per pair, and 45.1% of pairs exhibiting file-level overlap. As reference baselines, we evaluate a single agent that handles both PRs, as well as two multi-agent designs: assigning each PR to a separate agent and merging their patches afterwards, and letting a main agent delegate sub-tasks to sub-agents while it works. Even the best-performing configuration resolves only about one-fifth of tasks on average over three trials, compared with over 80% for state-of-the-art systems on SWE-Bench Verified, showing that naturally concurrent PR-composition tasks remain a largely unsaturated challenge for current coding agents.

1 INTRODUCTION

Coding agents powered by large language models (LLMs) take on software engineering tasks from function-level code generation (Chen et al., 2021) to entire programs (Yang et al., 2026). Progress is tracked largely through SWE-Bench (Jimenez et al., 2024) and its extensions, which turn real GitHub issues into tasks graded by the original developers' tests. This format shapes training and inference alike: models are fine-tuned on agent trajectories from executable repositories (Pan et al., 2025; Yang et al., 2025) and trained with reinforcement learning on issue resolution (Wei et al., 2025; team et al., 2025; Team et al., 2026); at inference time, agent scaffolds let models navigate, edit, and run code (Yang et al., 2024a; Wang et al., 2025), agentless pipelines run fixed localize, repair, and validate steps (Xia et al., 2024), and learned verifiers select among sampled patches (Pan et al., 2025). The best systems now resolve over 80% of SWE-Bench Verified (OpenAI, 2026b).

Despite these gains, existing benchmarks scope each task to a single developer's work (Figure 1). In practice, however, software engineering is inherently concurrent: multiple developers modify the same codebase simultaneously and coordinate through pull requests. Agents intensify the pattern (Johnston et al., 2026). 79.4% of agent PRs are open at the same time as another agent PR in the same repository (Xu et al., 2026). How coding agents handle concurrent changes remains an open question.

Recent work has produced collaborative benchmarks, but they construct the collaborative setting: CooperBench (Khatua et al., 2026) gives two agents a pair of features it constructs, and SyncMind (Guo et al., 2025) tests whether an agent recovers when its copy of the code falls out of sync with a collaborator's. Merge-conflict benchmarks (Schesch & Ernst, 2026) only start evaluation after conflicts have already been created. Critically, no existing benchmark scores whether agents

[1]Evaluation harness: `https://anonymous.4open.science/r/codev-bench-42AZ`

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.

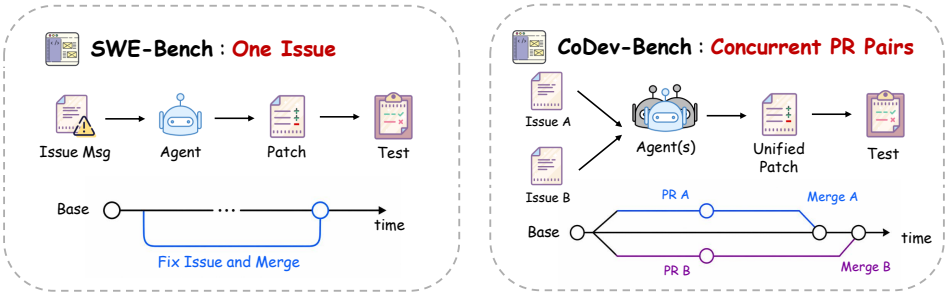


Figure 1: SWE-Bench evaluates a single agent on an isolated issue; CODEV-BENCH presents two concurrent PRs to one or more agents and evaluates against merged tests from both PRs.

can deliver both changes of a naturally concurrent pair, the upstream task from which such conflicts arise (Table 1).

In contrast, GitHub's historical data contains rich, naturally occurring concurrent signals: pairs of PRs that branch from the same base commit, undergo independent review, and are ultimately merged into the main branch. Such pairs carry the full complexity of concurrent development (Perry et al., 2001): each PR is written for its own goal, yet the two often depend on each other through the code they share, in ways neither PR description mentions (Mens, 2002; Brun et al., 2011; Gall et al., 1998). This motivates the central research question of this paper: *Can current coding agents simultaneously resolve two concurrent pull requests that share a codebase?* The task exposes what single-issue benchmarks cannot measure: whether an agent can implement one change while preserving the behavior another concurrent change requires.

To address this question, we introduce CODEV-BENCH: **206 naturally concurrent pull-request pairs** from **137 production-scale repositories** across **six languages** (Figure 2), with hidden tests audited (§3.2) to avoid a flaw OpenAI found in SWE-Bench Verified: tests that check implementation details rather than behavior (OpenAI, 2026b). We evaluate a single agent that handles both PRs against two multi-agent designs: assigning each PR to a separate agent and merging their patches afterwards, and letting a main agent delegate sub-tasks to sub-agents while it works.

Current agents fall well short on CODEV-BENCH, far below what frontier systems achieve on single-issue benchmarks. Yet many of the unresolved tasks miss by little: a wrong answer often fails just one hidden test. We identify two mechanisms behind these failures: agents' verification does not scale with the doubled change, and they overlook how the two changes depend on each other through shared code. A natural response is to split the work across agents, but neither of the multi-agent designs we evaluate helps consistently.

In summary, our contributions are:

- We construct CODEV-BENCH, the first benchmark for resolving naturally concurrent PR pairs, comprising 206 pairs drawn from 137 production-scale open-source repositories across six programming languages.
- Using CODEV-BENCH, we find that current agents fall far short: the best configuration resolves only 21.8% of pairs, and splitting the work across agents brings no consistent gain.
- We will publicly release the full benchmark data, containerized evaluation toolkit, and all agent trajectories with cost records to support future research on concurrent software development.

2 RELATED WORK

Single-task coding benchmarks. Function-level benchmarks (Chen et al., 2021; Austin et al., 2021; Hendrycks et al., 2021) are largely saturated, with extensions to code understanding and runtime-behavior reasoning (Manh et al., 2025; Hoang et al., 2026; Chen et al., 2024). SWE-Bench (Jimenez et al., 2024) shifted the field to verifiable patches on real GitHub issues, with a human-verified subset (OpenAI, 2024) and extensions across task scope and horizon (Le et al., 2026; Huang et al., 2026), language and modality coverage (Rashid et al., 2025; Zan et al., 2025; Yang et al., 2024b), data synthesis (Yang et al., 2025), agent training (Pan et al., 2025), temporal freshness (Zhang et al., 2025; Badertdinov et al., 2025), explore capability (Zhang et al., 2026c; Li

Table 1: Comparison with prior benchmarks. *Avg Scope*: non-test files and lines changed by the gold patch (for CODEV-BENCH, the two PRs of a pair combined). *Concurrent*: multiple agents or tasks operate on the same codebase. *Naturally Occurring*: concurrent structure comes from real developer activity, not synthetic construction. *Real File Overlap*: file overlap arises from independent developer choices. Numbers for prior work are as reported by the original papers.

Benchmark	# Repos	# Langs	Avg Scope (files / LOC)	*Properties*				
				Real Repo	Multi-task per Inst.	Concur-rent	Naturally Occurring	Real File Overlap
SWE-Bench	12	1	1.7 / 33	✓	✗	✗	✓	✗
Multi-SWE-Bench	39	7	–	✓	✗	✗	✓	✗
SWE-EVO	7	1	21 / 610	✓	✓	✗	✓	✗
SyncMind	21	1	–	✓	✓	✓	✗	✗
CooperBench	12	4	1.4 / 52	✓	✓	✓	✗	✗
Merge-Bench	1,439	11	1 conflict hunk	✓	✗	✓	✓	✓
CODEV-BENCH	**137**	**6**	**17.1 / 507**	✓	✓	✓	✓	✓

et al., 2026), enterprise scale (Deng et al., 2025), and secure coding (Chen et al., 2026a). Across all of these, every task is scoped to a single developer's work: one function, one issue, or one release.

Software engineering agents. Early protocols for LLM-driven repair (Xia & Zhang, 2023; Chen et al., 2023) grew into end-to-end autonomous coding agents (Cognition AI, 2024), which differ mainly in interface and pipeline design (Yang et al., 2024a; Wang et al., 2025; Zhang et al., 2024; Xia et al., 2024; Wang et al., 2024; Zhang et al., 2026b; Chen et al., 2026b); Mini-SWE-Agent (Yang et al., 2024a) distills the recipe into a few hundred lines of open-source code and serves as one of our harnesses. On the model side, SWE-RL (Wei et al., 2025), DeepSWE-Preview (Luo et al., 2025), CWM (team et al., 2025), and Kimi-K2 (Team et al., 2026) apply RL and code-trajectory fine-tuning to SE tasks, and behavioral studies examine agent trajectories beyond final-patch correctness (Nashid et al., 2026). All of these systems assume a single agent.

Multi-agent frameworks and collaborative benchmarks. Multi-agent designs for SE decompose the task through role specialization and sub-task partitioning (Hong et al., 2024; Qian et al., 2024; Islam et al., 2024; Zhang et al., 2026a; Huang et al., 2024; Shi et al., 2025; Nguyen et al., 2024; Li et al., 2025), differing mainly in the number of roles and their coordination protocol; Claude Code CLI (Anthropic, 2026a) and Copilot CLI (GitHub, 2026) implement a runtime variant of the same idea, spawning sub-agents on demand during a session.

Collaborative benchmarks each capture only part of concurrent development. In CooperBench (Khatua et al., 2026), two agents implement two features drawn from a pool the benchmark builds for each repository (1.4 files per task), whereas CODEV-BENCH pairs are PRs real developers wrote independently, touching 17.1 non-test files on average and sharing files in 45.1% of cases. Where SyncMind (Guo et al., 2025) builds out-of-sync states from 21 repositories to test recovery, CODEV-BENCH observes concurrency directly in the history of 137. Real conflicts do appear in Merge-Bench (Schesch & Ernst, 2026), but as hunks to resolve after both changes exist, not as whole PRs to write. Naturally occurring pairs, real file overlap, and PR-level scope meet only in CODEV-BENCH (Table 1).

3 THE CODEV-BENCH DATASET

3.1 TASK FORMULATION

Each CODEV-BENCH instance asks a coding agent for one unified patch that satisfies two naturally concurrent PRs on a shared codebase (Figure 3; Appendix G gives a full example). Formally, an instance is a tuple (R_b, s_A, s_B, T): the repository snapshot R_b at base commit b, problem statements s_A and s_B for the two PRs, and a hidden test suite T made up of the tests the two PRs add or modify, $T_A \cup T_B$, together with the repository's existing tests. A *solver*, single-agent, multi-agent, or otherwise, is any function $f(R_b, s_A, s_B) \mapsto p$ returning a unified patch p.

Evaluation. The patch p is applied to R_b and the full test suite T is executed; a run is **Resolved** if and only if every test in T passes. A patch that applies but fails at least one hidden test is a **Wrong**

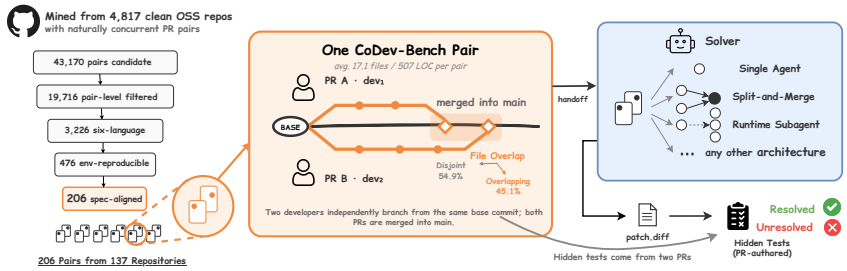


Figure 2: Overview of CODEV-BENCH. Left: the construction pipeline mines concurrent PR pairs from 4,817 repositories through progressive filtering and a behavioral-test audit. Center: each task presents two PRs branching from a shared base commit, with 45.1% exhibiting file-level overlap. Right: any solver architecture produces a unified patch evaluated against hidden PR-authored tests.

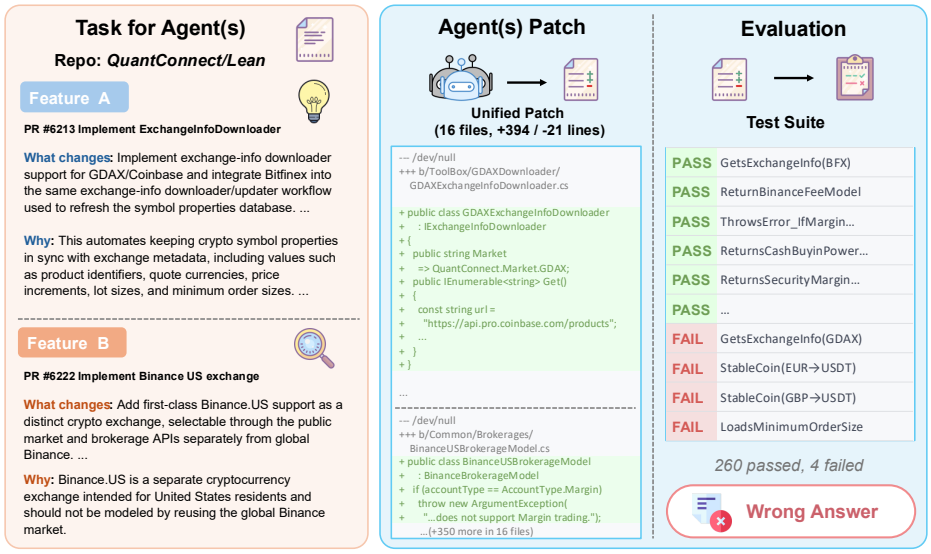


Figure 3: A CODEV-BENCH task (QuantConnect/Lean). Left: the problem statements of two PRs developed independently from the same base commit, one adding an exchange-info downloader and the other adding Binance.US support. Center: one or more agents produce a single unified patch that implements both features. Right: the patch is run against the hidden tests of both PRs, and the run counts as Resolved only if all of them pass.

Answer; one under which the suite cannot run to completion, typically because the hidden tests no longer compile against the patched code, is a **Crash**. Both count as failures.

File overlap. Let g_A and g_B be the gold patches of the two PRs, and $\text{files}(g)$ the set of files modified by g. A pair is *overlapping* if $\text{files}(g_A) \cap \text{files}(g_B) \neq \emptyset$, and *disjoint* otherwise. File overlap is a standard measure of parallel development (Perry et al., 2001), but not a measure of coupling: changes that merge cleanly can still break each other's builds or tests (Brun et al., 2011). We use it to describe the dataset and, in §4.3, to test whether overlapping pairs are harder.

3.2 BENCHMARK CONSTRUCTION

Construction proceeds in three stages. From a pool of 4,817 open-source repositories we mine 43,170 candidate *PR pairs*. Each pair consists of two PRs that branch from the same base commit, are authored by different contributors, and are both eventually merged into main. Pair-level filters (size, bot activity, temporal overlap, a six-language whitelist) reduce the pool to 3,226. Each survivor is then validated in a reproducible container (its hidden tests must fail before and pass after the gold patch, and skipping either PR's changes must fail) and audited so that the hidden tests assert behavior rather than implementation details. Without this audit, a correct patch written differently

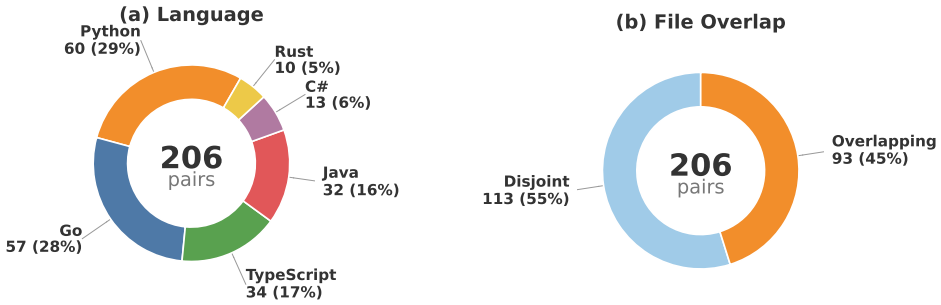


Figure 4: Distributional properties of CODEV-BENCH (206 pairs from 137 repositories). (a) Primary language of the repository. (b) File overlap: whether the two PRs of a pair modify at least one common file (overlapping) or none (disjoint).

from the gold patch could fail the hidden tests: OpenAI found such tests in 35.5% of 138 hard SWE-Bench Verified tasks it audited (OpenAI, 2026b). Each remaining pair receives a problem statement describing the capability each PR adds without revealing its internal implementation; a final manual audit leaves **206** pairs. Filter criteria and audit protocol are in Appendices B and C; Figure 2 (left) shows the funnel.

3.3 FEATURES OF CODEV-BENCH

Together, the two PRs of a pair change 17.1 non-test files and 507 lines on average (median 13 files / 320 lines). The 206 pairs come from 137 repositories (median 11.5k GitHub stars) in six languages (Figure 4a). In 45.1% of pairs the two PRs share at least one file (1.90 on average; Figure 4b), so CODEV-BENCH is neither trivially disjoint nor dominated by overlap as synthetic collaboration benchmarks are. Feature+feature (34.5%) and bugfix+feature (20.4%) are the most common pair types; further statistics are in Appendix B, contamination checks in Appendix F.

4 EXPERIMENTS

4.1 EXPERIMENTAL SETUP

Agents. We use three representative agent frameworks. Mini-SWE-Agent (Yang et al., 2024a) is a lightweight, community-adopted ReAct-style agent; we adopt it as the primary framework because its minimal architecture simplifies reproduction and trajectory audit. Claude Code CLI (Anthropic, 2026a) and Copilot CLI (GitHub, 2026) are production-grade coding agents from Anthropic and GitHub respectively; both natively support runtime sub-agent decomposition via a built-in agent-dispatch tool.

Approaches. On each framework we evaluate a single-agent approach and one or two multi-agent approaches, giving seven configurations per model; the names below are the row labels of Table 2.

- **Single Agent** (all three frameworks): one agent receives both problem statements and produces the unified patch.
- **Split-and-merge** (Mini-SWE-Agent): two workers each implement one PR independently, and their patches are combined in one of two ways: **Git Merge**, where git applies the two patches in turn without any model call (details in Appendix F), or **LLM Merge**, where an LLM Merger produces the unified patch.
- **Sub-Agent** (the two CLIs): the main agent dispatches sub-agents through the framework's built-in agent-dispatch tool and integrates their results.

Models. We evaluate four frontier LLMs: Claude Opus 4.8 (Anthropic, 2026b), Grok 4.5 (xAI, 2026), GPT-5.6-Sol (OpenAI, 2026a), and the open-weight Kimi K2.6 (Kimi Team, 2026), giving 28 configurations, each run three times on all 206 pairs. Table 2 reports three-trial means (per-trial results in Appendix D); trajectory analyses use the first trial. All runs share a container with network isolation and git-history stripping (Appendix F).

Table 2: Main results (206 pairs, mean over three trials). Resolved = Resolved Rate (%); Cost = mean USD per pair. Bold marks the highest Resolved Rate per model. Gains over Single Agent are small and not significant within any trial except one (1 of 48 paired comparisons at $p < 0.05$; Table 7). Standard deviations in Appendix D.

Framework	Approach	Opus 4.8		Grok 4.5		GPT-5.6-Sol		Kimi K2.6	
		Resolved	Cost	Resolved	Cost	Resolved	Cost	Resolved	Cost
Mini-SWE-Agent	Single Agent	20.9	$3.3	18.6	$2.4	15.7	$1.1	12.0	$2.6
	Git Merge	20.7	$3.5	19.7	$3.1	16.2	$1.4	14.2	$3.1
	LLM Merge	21.4	$3.6	**21.4**	$3.2	**16.5**	$1.5	**14.4**	$3.1
Claude Code CLI	Single Agent	20.6	$5.3	21.0	$2.9	14.6	$7.2	12.1	$3.6
	Sub-Agent	21.7	$5.7	19.1	$5.9	13.8	$15.1	11.7	$3.9
Copilot CLI	Single Agent	21.7	$4.6	18.6	$2.9	14.6	$2.0	13.4	$2.7
	Sub-Agent	**21.8**	$5.8	18.9	$4.7	13.9	$3.1	12.1	$2.5

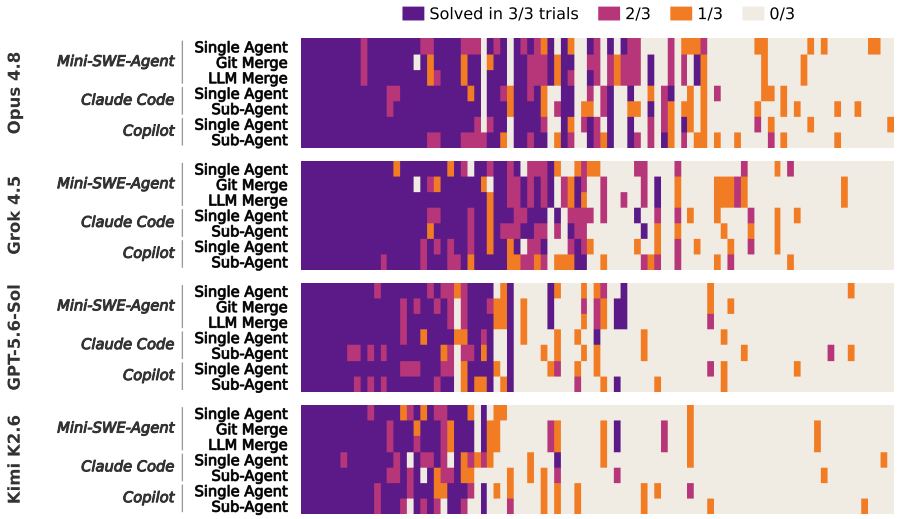


Figure 5: Per-pair outcomes of all 28 configurations over three trials. Rows: configurations grouped by model and framework. Columns: the 89 pairs solved at least once, ordered by total solves; shading gives the number of trials (out of three) in which the configuration solves the pair. The other 117 pairs (57%), which no configuration solves in any trial, are omitted.

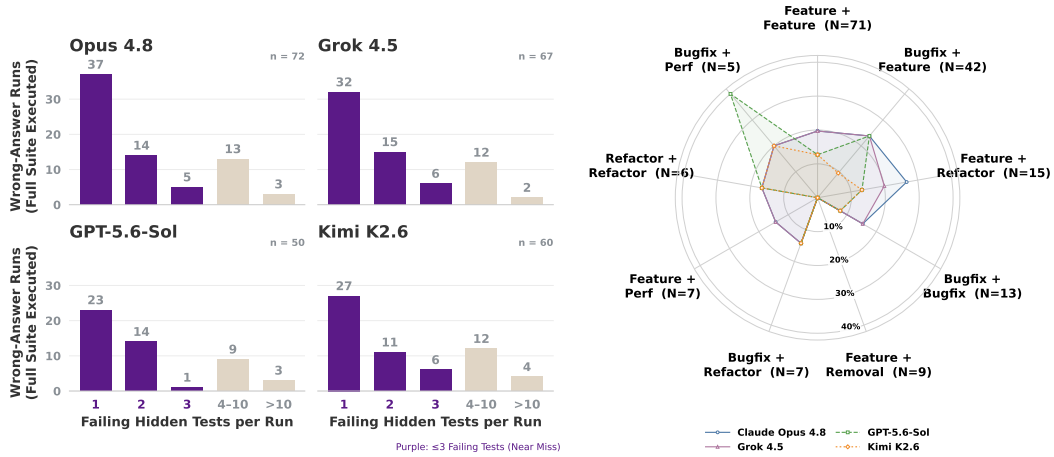
Metrics. We report the **Resolved Rate**, the share of the 206 pairs whose run is Resolved (§3.1), and the **Average Cost**, the mean USD spent per pair, to compare efficiency across solver architectures. Costs are computed from per-request token usage at the rates in Table 8.

4.2 How Hard Are Concurrent Tasks?

4.2.1 Main Results

Table 2 reports Resolved Rates over the full matrix. The benchmark poses a substantial challenge to current agents. The best-performing configuration reaches only 21.8% (Opus × Copilot Sub-Agent; Mini-SWE-Agent Single Agent reaches 20.9% for Opus), compared with over 80% for state-of-the-art systems on SWE-Bench Verified (OpenAI, 2026b), whose single-issue tasks are an order of magnitude smaller in scope.

Despite the low rates, the benchmark remains discriminative and far from saturation: 57% of the pairs are solved by no configuration in any of the three trials, and pooling all 28 configurations and all trials reaches only 43.2% (Figure 5). Difficulty is a stable property of the tasks rather than noise: 73.3% of pairs are failed by all four models and 9.7% are solved by all four, with Fleiss' $\kappa = 0.674$ across models (Fleiss, 1971) (Appendix D.1). Performance varies sharply across task dimensions. By language, Java and C# are systematically low (C# zero in 15 of the 28 configurations), while TypeScript and Go are the highest. By task combination (the types of the two PRs), pairs that



(a) Distribution of failed-test counts among wrong answers that executed all the tests (Mini-SWE-Agent Single Agent, first trial); all four models peak at "exactly 1".

(b) Resolved Rate by task combination × model (Mini-SWE-Agent Single Agent, first trial, combinations with N≥5); all models collapse to zero on feature+removal.

Figure 6: The last-mile phenomenon and task-type difficulty.

Table 3: Failure attribution for Grok 4.5 × Mini-SWE-Agent Single Agent (165 failures). One side / Both: Single Agent's patch fails one side's hidden tests or both; 23 inseparable failures cannot be scored per side. Parentheses (incapacity row): Single Agent's patch fails exactly the same sides as the split-and-merge worker given that PR alone.

Category	Count	Share	One side	Both
Inseparable	96	58.2%	16	57
Single-task incapacity	65	39.4%	30 (27)	35 (26)
Separable	4	2.4%	3	1
Total failures	165	100%	49	93

combine a feature addition with a removal are the hardest: **no Mini-SWE-Agent Single Agent run solves any of the 9 such pairs**, and only 2 of the 9 are solved by any configuration (Figure 6, right).

4.2.2 The Last-Mile Phenomenon

Between half and two-thirds of wrong answers execute the full test suite; the rest are truncated by fail-fast scripts. Among those that do, nearly half fail exactly one hidden test and 73–79% fail three or fewer (Figure 6, left). The median distance from success is 1 to 2 failed tests for every model.

4.2.3 Failure Attribution

Most failures are not ones that splitting the work could fix. In a *per-side evaluation*, each split-and-merge worker receives only one PR, and we run that PR's hidden tests alone on the worker's patch (validation in Appendix D). Each failure falls into one of three categories. **Inseparable**: at least one side cannot be scored alone, usually because its gold patch fails its own tests without the other side (Appendix D.1). **Single-task incapacity**: both sides can be scored, but the model fails one even when given it alone. **Separable**: the model solves each PR alone but not both together.

We run the attribution on Grok 4.5 as a case study. Of its 165 failures under Mini-SWE-Agent Single Agent (Table 3), only 4 are separable. Scoring Single Agent's patch against each side's tests (right-hand columns of Table 3) agrees with the categories: in 53 of the 65 incapacity failures, it fails exactly the same sides as the split-and-merge worker given that PR alone. Of the 142 failures scorable on both sides, only 49 fail a single side (22 pass only side A, 27 only side B), so most failures are not a matter of finishing one PR and dropping the other. The 93 two-side failures are an upper bound, as shared tests and build errors count against both.

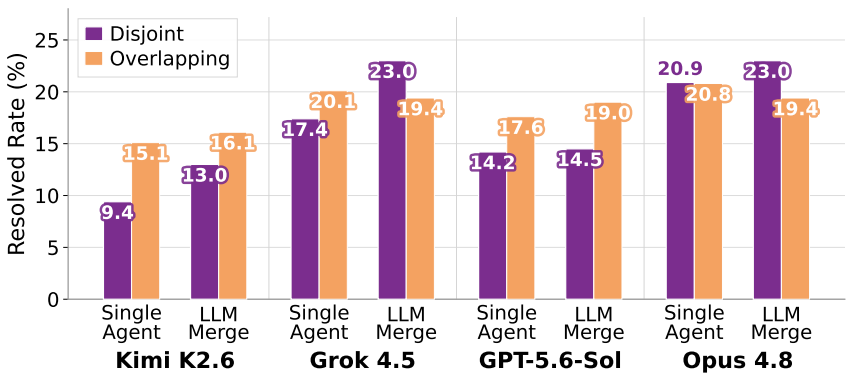


Figure 7: Resolved Rate stratified by file overlap on Mini-SWE-Agent (Single Agent and LLM Merge), averaged over three trials. No significant difference between groups for any of the four models.

With so few separable failures, giving each PR to its own worker should rescue few, and it does. Git Merge, which simply applies both workers' patches, solves only 8 of the 165 pairs Single Agent fails (4.8%): all 4 separable ones, plus 3 inseparable and 1 incapacity pair. Across the four models, it rescues 2.3–7.3% of Single Agent's failures (trajectory-level profiles in Appendix E).

4.3 Sources of Concurrent Difficulty

Simpler explanations do not account for these failures. File overlap does not predict difficulty: disjoint and overlapping pairs show no significant difference in Resolved Rate for any model (Figure 7; permutation tests in Appendix D.2). Agents also do not simply skip one of the two PRs: 97–99% of trajectories make substantive edits on both sides, and runs that edit only one side account for 1.0–3.4%. Added volume alone does not explain which pairs fail either (Appendix D.2).

The difficulty instead takes two forms in the concurrent setting: verification that does not scale with the doubled change, and implicit semantic coupling between the two changes.

4.3.1 Verification Gaps and Semantic Coupling

Mechanism 1: Verification does not scale with the change. A PR pair roughly doubles the editing scope, from 8.9 non-test files per side to 17.1 combined, but agents do not expand their verification to match. As the combined gold change grows, so does the number of files an agent edits, at a Spearman correlation of 0.40 to 0.65 across the four models; the number of test executions does not, at -0.09 to 0.01. The share of runs that never execute a test rises from 19–25% on pairs of at most 10 files to 32–50% at 21–40, and only 2–8% of Mini-SWE-Agent Single Agent runs ever run the repository's full test suite before submitting (Appendix E). Two cases there show two ways a run's checks fail to reach the combined change. In the first, the checks do not reach the full change: the agent compiles successfully yet never runs the test suite, submitting with ample budget remaining. In the second, the checks encode the same wrong assumption as the code, so every self-test passes while hidden tests fail.

Mechanism 2: Implicit semantic coupling. Concurrent PRs often touch shared abstractions, invariants, or interfaces, so the two changes depend on each other. This is common in CoDevBench: in 83 of the 206 pairs, the hidden tests on one PR's side pass only when the other PR's changes are also applied (Appendix D.1). File overlap does not reveal this dependency: in 41 of these 83 pairs, the two PRs share no file. The dependency is invisible in either diff: on nginx/kubernetes-ingress, PR A's own hidden tests compare the whole generated NGINX configuration, so they encode the default `Host` header that only PR B adds, and fail until both changes are present (Appendix E).

The two mechanisms compound. Neither problem statement nor either diff shows how the two changes interact. Coupling means an agent has to check it anyway, and it is exactly what verification that does not scale leaves out: only 2–8% of runs ever run the full test suite, which is what covers both changes at once. §4.4 asks whether splitting the work helps.

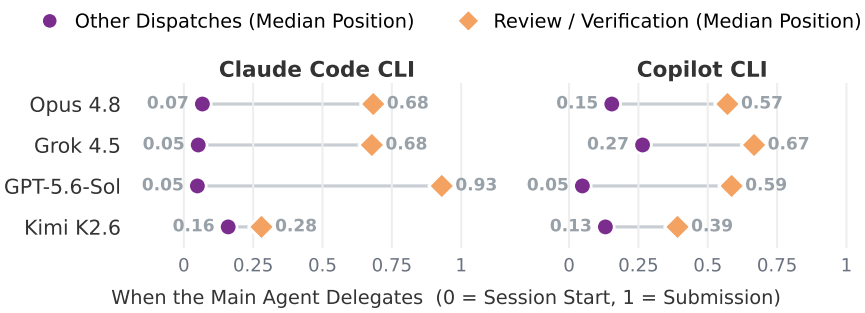


Figure 8: Median position of review/verification (orange) and all other (purple) dispatches, i.e., how far into the main agent's turns each is made, for the 8 sub-agent configurations in the first trial (left: Claude Code; right: Copilot). Review comes later in every configuration.

4.4 Can Decomposition Help?

4.4.1 No Consistent Gain

Neither decomposition style yields a reliable gain (Table 2). On Mini-SWE-Agent, LLM Merge is ahead of Single Agent for all four models, but by at most 7 of 206 pairs in any trial and at \$0.3–0.8 more per pair; on the two CLIs, Sub-Agent stays within 2 points of Single Agent, usually at higher cost. Only 1 of the 48 paired comparisons reaches $p < 0.05$, and the small net differences hide pairs gained and lost in every trial (Table 7).

Nor does either style save wall-clock time. On Mini-SWE-Agent, even with its two workers running in parallel and only the merger added after them, the split is level with Single Agent; on the two CLIs, the sub-agent mode is slower in 7 of the 8 comparisons (Appendix D.3).

4.4.2 How Sub-Agents Are Used

In the CLIs' runtime sub-agent mode, verification is never divided across workers. Across models and frameworks, the main agent hands a task to a sub-agent 6,356 times: exploration accounts for 74.1–86.5% of Copilot dispatches and review for 18.2–49.4% of Claude Code's, under the labelling described in Appendix D. What review there is comes late: its median dispatch falls 28–93% of the way through the main agent's turns, against 5–27% for other dispatches (Figure 8). Late reports often get no follow-up: when a sub-agent reports back in the last tenth of a run, the main agent makes no further edit in 45–87% of cases, whether the report called for no change or went unheeded. Sub-agents therefore do not supply the verification that Mechanism 1 finds missing.

5 Conclusion

We introduce CoDev-Bench, a benchmark of naturally concurrent PR-pair tasks from 137 open-source repositories across six languages. The strongest configuration resolves only 21.8% of tasks, yet many wrong answers fail just one hidden test. Neither file overlap nor the added volume of change explains this difficulty. It takes two forms: verification that does not scale with the doubled change, and the way the two changes interact through shared code. Splitting the work across agents does not help consistently. CoDev-Bench thus provides a testbed for improving both agent harnesses and the models behind them. For harness design, verification is the part to manage: no current workflow plans a full test run, review sub-agents are called late, and no framework divides the test suite across workers. A workflow that splits the work needs to know which pairs are coupled, and file overlap does not tell. For models, this points to a concrete capability to improve: recognizing when two changes are coupled, that is, when two individually correct changes can still break each other through shared code. A model that can judge this would know both when a task is safe to split and what to check after combining the two changes. Limitations are discussed in Appendix A.

ETHICS STATEMENT

All data in CODEV-BENCH derive from publicly available open-source repositories with permissive licenses (Apache-2.0, MIT, BSD). The benchmark manifest and problem statements contain no personally identifiable information; contributor identities are used only as a boolean same-author filter during pair mining and are not released. Our evaluation involves no human subjects. The benchmark is designed to measure and expose limitations of current coding agents, not to amplify their capabilities for adversarial purposes; we believe transparent reporting of failure modes serves the community's interest in building safer and more reliable AI-assisted software engineering.

REPRODUCIBILITY STATEMENT

We take several steps to support reproducibility. The full benchmark data—the 206-pair manifest with metadata, frozen Docker container images, per-pair evaluation kits with hidden tests, and exact agent invocations—will be released with the paper (Appendix H). CLI versions are pinned (Claude Code 2.1.251, Copilot CLI 1.0.74), thinking settings are fixed per model, and resource limits are uniform across all configurations (Appendix F). All reported costs are computed from raw per-request token counts at list prices (Table 8), not from framework-reported totals. The construction pipeline's filter criteria, audit protocol, and attrition statistics are detailed in Appendices B and C. Per-side evaluation validity checks are described in Appendix D. All agent and evaluation trajectories, including per-run cost records, will be released with them.

AI USE STATEMENT

In this work, we used generative AI tools for synthetic data generation, qualitative analysis, and method implementation: each pair's problem statements are written by a three-pass LLM pipeline from the merged gold diff; each pair's hidden tests are labelled for behavioral-versus-implementation focus by an LLM judge (Appendix C); and AI coding assistants were used while writing the evaluation harness and the analysis scripts. We have not used generative AI tools to propose the hypotheses of this work, to design the experiments, or for translation, and no model judges run outcomes: every run is scored by executing the hidden tests and parsing their output. Developing theoretical models and formulating or proving mathematical claims are not applicable to this work. Additionally, we used generative AI tools for figure generation and for drafting and editing the manuscript's prose. We have reviewed all AI-assisted work: every generated problem statement and every hidden-test label was read and corrected by the authors before entering the manifest (Appendix C), and all analyses, numbers, and claims in the paper were recomputed and checked by the authors. We take responsibility for the final content of this work, including text, claims or artifacts produced with the aid of generative AI.

REFERENCES

Anthropic. Claude code. https://claude.com/product/claude-code, 2026a. Accessed: 2026-09-10.

Anthropic. System card: Claude Opus 4.8. https://www.anthropic.com/claude-opus-4-8-system-card, 2026b.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. Program synthesis with large language models, 2021. URL https://arxiv.org/abs/2108.07732.

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, 2025. URL https://arxiv.org/abs/2505.20411.

Yuriy Brun, Reid Holmes, Michael D. Ernst, and David Notkin. Proactive detection of collaboration conflicts. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, pp. 168–178, New

York, NY, USA, 2011. Association for Computing Machinery. ISBN 9781450304436. doi: 10.1145/2025113.2025139. URL https://doi.org/10.1145/2025113.2025139.

Junkai Chen, Zhiyuan Pan, Xing Hu, Zhenhao Li, Ge Li, and Xin Xia. Reasoning runtime behavior of a program with llm: How far are we?, 2024. URL https://arxiv.org/abs/2403.16437.

Junkai Chen, Huihui Huang, Yunbo Lyu, Junwen An, Jieke Shi, Chengran Yang, Ting Zhang, Haoye Tian, Yikun Li, Zhenhao Li, Xin Zhou, Xing Hu, and David Lo. Securevibebench: Benchmarking secure vibe coding of ai agents via reconstructing vulnerability-introducing scenarios, 2026a. URL https://arxiv.org/abs/2509.22097.

Junkai Chen, Chengran Yang, Xing Hu, Zhenhao Li, Xin Xia, and David Lo. Agentexecutor: Partial code execution via agentic context generation, 2026b. URL https://arxiv.org/abs/2608.05959.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code, 2021.

Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. Teaching large language models to self-debug, 2023. URL https://arxiv.org/abs/2304.05128.

Cognition AI. Introducing Devin, the first AI software engineer. https://www.cognition.ai/blog/introducing-devin, March 2024. Accessed: 2026-07-19.

Conventional Commits. Conventional commits 1.0.0. https://www.conventionalcommits.org/en/v1.0.0/, 2019.

Xiang Deng, Jeff Da, Edwin Pan, Yannis Yiming He, Charles Ide, Kanak Garg, Niklas Lauffer, Andrew Park, Nitin Pasari, Chetan Rane, Karmini Sampath, Maya Krishnan, Srivatsa Kundurthy, Sean Hendryx, Zifan Wang, Vijay Bharadwaj, Jeff Holm, Raja Aluri, Chen Bo Calvin Zhang, Noah Jacobson, Bing Liu, and Brad Kenstler. Swe-bench pro: Can ai agents solve long-horizon software engineering tasks?, 2025. URL https://arxiv.org/abs/2509.16941.

Joseph L. Fleiss. Measuring nominal scale agreement among many raters. *Psychological Bulletin*, 76(5):378–382, November 1971. ISSN 0033-2909. doi: 10.1037/h0031619. URL http://dx.doi.org/10.1037/h0031619.

H. Gall, K. Hajek, and M. Jazayeri. Detection of logical coupling based on product release history. In *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)*, pp. 190–198, 1998. doi: 10.1109/ICSM.1998.738508.

GitHub. Github copilot cli. https://github.com/features/copilot/cli/, 2026. Accessed: 2026-09-10.

Xuehang Guo, Xingyao Wang, Yangyi Chen, Sha Li, Chi Han, Manling Li, and Heng Ji. Syncmind: Measuring agent out-of-sync recovery in collaborative software engineering, 2025. URL https://arxiv.org/abs/2502.06994.

Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and Jacob Steinhardt. Measuring coding challenge competence with apps. *NeurIPS*, 2021.

Anh Nguyen Hoang, Minh Le-Anh, Bach Le, and Nghi D. Q. Bui. Codewiki: Evaluating ai's ability to generate holistic documentation for large-scale codebases, 2026. URL https://arxiv.org/abs/2510.24428.

Sirui Hong, Mingchen Zhuge, Jiaqi Chen, Xiawu Zheng, Yuheng Cheng, Ceyao Zhang, Jinlin Wang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, Chenyu Ran, Lingfeng Xiao, Chenglin Wu, and Jürgen Schmidhuber. Metagpt: Meta programming for a multi-agent collaborative framework, 2024. URL https://arxiv.org/abs/2308.00352.

Dong Huang, Jie M. Zhang, Michael Luck, Qingwen Bu, Yuhao Qing, and Heming Cui. Agentcoder: Multi-agent-based code generation with iterative testing and optimisation, 2024. URL https://arxiv.org/abs/2312.13010.

Wenqi Huang, Charley Lee, Leonard Tng, and Serena Ge. Deepswe: Measuring frontier coding agents on original, long-horizon engineering tasks, 2026. URL https://arxiv.org/abs/2607.07946.

Md. Ashraful Islam, Mohammed Eunus Ali, and Md Rizwan Parvez. Mapcoder: Multi-agent code generation for competitive problem solving, 2024. URL https://arxiv.org/abs/2405.11403.

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?, 2024. URL https://arxiv.org/abs/2310.06770.

Drew Johnston, David Holtz, Alex Martin Richmond, Christopher Ong, Prasanna Tambe, and Aaron Chatterji. The shift to agentic ai: Evidence from codex, 2026. URL https://arxiv.org/abs/2606.26959.

Arpandeep Khatua, Hao Zhu, Peter Tran, Arya Prabhudesai, Frederic Sadrieh, Johann K. Lieberwirth, Xinkai Yu, Yicheng Fu, Michael J. Ryan, Jiaxin Pei, and Diyi Yang. Cooperbench: Why coding agents cannot be your teammates yet, 2026. URL https://arxiv.org/abs/2601.13295.

Kimi Team. Kimi K2.6: Advancing open-source coding. https://www.kimi.com/blog/kimi-k2-6, 2026.

J. Richard Landis and Gary G. Koch. The measurement of observer agreement for categorical data. *Biometrics*, 33(1):159–174, March 1977. ISSN 0006-341X. doi: 10.2307/2529310. URL http://dx.doi.org/10.2307/2529310.

Tue Le, Minh V. T. Thai, Dung Nguyen Manh, Huy Phan Nhat, and Nghi D. Q. Bui. Swe-evo: Benchmarking coding agents in long-horizon software evolution scenarios, 2026. URL https://arxiv.org/abs/2512.18470.

Han Li, Yuling Shi, Shaoxin Lin, Xiaodong Gu, Heng Lian, Xin Wang, Yantao Jia, Tao Huang, and Qianxiang Wang. Swe-debate: Competitive multi-agent debate for software issue resolution, 2025. URL https://arxiv.org/abs/2507.23348.

Han Li, Letian Zhu, Bohan Zhang, Rili Feng, Jiaming Wang, Yue Pan, Earl T. Barr, Federica Sarro, Zhaoyang Chu, and He Ye. Contextbench: A benchmark for context retrieval in coding agents, 2026. URL https://arxiv.org/abs/2602.05892.

Michael Luo, Naman Jain, Jaskirat Singh, Sijun Tan, Ameen Patel, Qingyang Wu, Alpay Ariyak, Colin Cai, Tarun Venkat, Shang Zhu, Ben Athiwaratkun, Manan Roongta, Ce Zhang, Li Erran Li, Raluca Ada Popa, Koushik Sen, and Ion Stoica. Deepswe: Training a fully open-sourced, state-of-the-art coding agent by scaling rl. Together AI Blog, 2025. URL https://www.together.ai/blog/deepswe.

Dung Nguyen Manh, Thang Phan Chau, Nam Le Hai, Thong T. Doan, Nam V. Nguyen, Quang Pham, and Nghi D. Q. Bui. Codemmlu: A multi-task benchmark for assessing code understanding & reasoning capabilities of codellms, 2025. URL https://arxiv.org/abs/2410.01999.

T. Mens. A state-of-the-art survey on software merging. *IEEE Transactions on Software Engineering*, 28(5):449–462, 2002. doi: 10.1109/TSE.2002.1000449.

Noor Nashid, Daniel Ding, Keheliya Gallaba, Ahmed E. Hassan, and Ali Mesbah. Beyond accuracy: Behavioral dynamics of agentic multi-hunk repair, 2026. URL https://arxiv.org/abs/2511.11012.

Minh Huynh Nguyen, Thang Phan Chau, Phong X. Nguyen, and Nghi D. Q. Bui. Agilecoder: Dynamic collaborative agents for software development based on agile methodology, 2024. URL https://arxiv.org/abs/2406.11912.

OpenAI. Introducing SWE-bench verified. https://openai.com/index/introducing-swe-bench-verified/, 2024.

OpenAI. GPT-5.6 system card. https://deploymentsafety.openai.com/gpt-5-6, 2026a.

OpenAI. Why SWE-bench verified no longer measures frontier coding capabilities. https://openai.com/index/why-we-no-longer-evaluate-swe-bench-verified/, 2026b. Accessed: 2026-09-24.

Jiayi Pan, Xingyao Wang, Graham Neubig, Navdeep Jaitly, Heng Ji, Alane Suhr, and Yizhe Zhang. Training software engineering agents and verifiers with swe-gym, 2025. URL https://arxiv.org/abs/2412.21139.

Dewayne E. Perry, Harvey P. Siy, and Lawrence G. Votta. Parallel changes in large-scale software development: an observational case study. *ACM Trans. Softw. Eng. Methodol.*, 10(3):308–337, July 2001. ISSN 1049-331X. doi: 10.1145/383876.383878. URL https://doi.org/10.1145/383876.383878.

Chen Qian, Wei Liu, Hongzhang Liu, Nuo Chen, Yufan Dang, Jiahao Li, Cheng Yang, Weize Chen, Yusheng Su, Xin Cong, Juyuan Xu, Dahai Li, Zhiyuan Liu, and Maosong Sun. Chatdev: Communicative agents for software development, 2024. URL https://arxiv.org/abs/2307.07924.

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

Benedikt Schesch and Michael D. Ernst. Merge-bench: Resolve merge conflicts with large language models, 2026. URL https://arxiv.org/abs/2605.25890.

Yuling Shi, Songsong Wang, Chengcheng Wan, Min Wang, and Xiaodong Gu. From code to correctness: Closing the last mile of code generation with hierarchical debugging, 2025. URL https://arxiv.org/abs/2410.01215.

FAIR CodeGen team, Jade Copet, Quentin Carbonneaux, Gal Cohen, Jonas Gehring, Jacob Kahn, Jannik Kossen, Felix Kreuk, Emily McMilin, Michel Meyer, Yuxiang Wei, David Zhang, Kunhao Zheng, Jordi Armengol-Estapé, Pedram Bashiri, Maximilian Beck, Pierre Chambon, Abhishek Charnalia, Chris Cummins, Juliette Decugis, Zacharias V. Fisches, François Fleuret, Fabian Gloeckle, Alex Gu, Michael Hassid, Daniel Haziza, Badr Youbi Idrissi, Christian Keller, Rahul Kindi, Hugh Leather, Gallil Maimon, Aram Markosyan, Francisco Massa, Pierre-Emmanuel Mazaré, Vegard Mella, Naila Murray, Keyur Muzumdar, Peter O'Hearn, Matteo Pagliardini, Dmitrii Pedchenko, Tal Remez, Volker Seeker, Marco Selvi, Oren Sultan, Sida Wang, Luca Wehrstedt, Ori Yoran, Lingming Zhang, Taco Cohen, Yossi Adi, and Gabriel Synnaeve. Cwm: An open-weights llm for research on code generation with world models, 2025. URL https://arxiv.org/abs/2510.02387.

Kimi Team, Yifan Bai, Yiping Bao, Y. Charles, Cheng Chen, Guanduo Chen, Haiting Chen, Huarong Chen, Jiahao Chen, Ningxin Chen, Ruijue Chen, Yanru Chen, Yuankun Chen, Yutian Chen, Zhuofu Chen, Jialei Cui, Hao Ding, Mengnan Dong, Angang Du, Chenzhuang Du, Dikang Du,

Yulun Du, Yu Fan, Yichen Feng, Kelin Fu, Bofei Gao, Chenxiao Gao, Hongcheng Gao, Peizhong Gao, Tong Gao, Yuyao Ge, Shangyi Geng, Qizheng Gu, Xinran Gu, Longyu Guan, Haiqing Guo, Jianhang Guo, Xiaoru Hao, Tianhong He, Weiran He, Wenyang He, Yunjia He, Chao Hong, Hao Hu, Yangyang Hu, Zhenxing Hu, Weixiao Huang, Zhiqi Huang, Zihao Huang, Tao Jiang, Zhejun Jiang, Xinyi Jin, Yongsheng Kang, Guokun Lai, Cheng Li, Fang Li, Haoyang Li, Ming Li, Wentao Li, Yang Li, Yanhao Li, Yiwei Li, Zhaowei Li, Zheming Li, Hongzhan Lin, Xiaohan Lin, Zongyu Lin, Chengyin Liu, Chenyu Liu, Hongzhang Liu, Jingyuan Liu, Junqi Liu, Liang Liu, Shaowei Liu, T. Y. Liu, Tianwei Liu, Weizhou Liu, Yangyang Liu, Yibo Liu, Yiping Liu, Yue Liu, Zhengying Liu, Enzhe Lu, Haoyu Lu, Lijun Lu, Yashuo Luo, Shengling Ma, Xinyu Ma, Yingwei Ma, Shaoguang Mao, Jie Mei, Xin Men, Yibo Miao, Siyuan Pan, Yebo Peng, Ruoyu Qin, Zeyu Qin, Bowen Qu, Zeyu Shang, Lidong Shi, Shengyuan Shi, Feifan Song, Jianlin Su, Zhengyuan Su, Lin Sui, Xinjie Sun, Flood Sung, Yunpeng Tai, Heyi Tang, Jiawen Tao, Qifeng Teng, Chaoran Tian, Chensi Wang, Dinglu Wang, Feng Wang, Hailong Wang, Haiming Wang, Jianzhou Wang, Jiaxing Wang, Jinhong Wang, Shengjie Wang, Shuyi Wang, Si Wang, Xinyuan Wang, Yao Wang, Yejie Wang, Yiqin Wang, Yuxin Wang, Yuzhi Wang, Zhaoji Wang, Zhengtao Wang, Zhengtao Wang, Zhexu Wang, Chu Wei, Qianqian Wei, Haoning Wu, Wenhao Wu, Xingzhe Wu, Yuxin Wu, Chenjun Xiao, Jin Xie, Xiaotong Xie, Weimin Xiong, Boyu Xu, Jinjing Xu, L. H. Xu, Lin Xu, Suting Xu, Weixin Xu, Xinran Xu, Yangchuan Xu, Ziyao Xu, Jing Xu, Jing Xu, Junjie Yan, Yuzi Yan, Hao Yang, Xiaofei Yang, Yi Yang, Ying Yang, Zhen Yang, Zhilin Yang, Zonghan Yang, Haotian Yao, Xingcheng Yao, Wenjie Ye, Zhuorui Ye, Bohong Yin, Longhui Yu, Enming Yuan, Hongbang Yuan, Mengjie Yuan, Siyu Yuan, Haobing Zhan, Dehao Zhang, Hao Zhang, Wanlu Zhang, Xiaobin Zhang, Yadong Zhang, Yangkun Zhang, Yichi Zhang, Yizhi Zhang, Yongting Zhang, Yu Zhang, Yutao Zhang, Yutong Zhang, Zheng Zhang, Haotian Zhao, Yikai Zhao, Zijia Zhao, Huabin Zheng, Shaojie Zheng, Longguang Zhong, Jianren Zhou, Xinyu Zhou, Zaida Zhou, Jinguo Zhu, Zhen Zhu, Weiyu Zhuang, and Xinxing Zu. Kimi k2: Open agentic intelligence, 2026. URL https://arxiv.org/abs/2507.20534.

Xingyao Wang, Yangyi Chen, Lifan Yuan, Yizhe Zhang, Yunzhu Li, Hao Peng, and Heng Ji. Executable code actions elicit better llm agents, 2024. URL https://arxiv.org/abs/2402.01030.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, Hoang H. Tran, Fuqiang Li, Ren Ma, Mingzhang Zheng, Bill Qian, Yanjun Shao, Niklas Muennighoff, Yizhe Zhang, Binyuan Hui, Junyang Lin, Robert Brennan, Hao Peng, Heng Ji, and Graham Neubig. Openhands: An open platform for ai software developers as generalist agents, 2025. URL https://arxiv.org/abs/2407.16741.

Yuxiang Wei, Olivier Duchenne, Jade Copet, Quentin Carbonneaux, Lingming Zhang, Daniel Fried, Gabriel Synnaeve, Rishabh Singh, and Sida I. Wang. Swe-rl: Advancing llm reasoning via reinforcement learning on open software evolution, 2025. URL https://arxiv.org/abs/2502.18449.

xAI. Introducing Grok 4.5. https://x.ai/news/grok-4-5, 2026.

Chunqiu Steven Xia and Lingming Zhang. Conversational automated program repair, 2023. URL https://arxiv.org/abs/2301.13246.

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

George Xu, Arjun Subramanian, and Nithilan Karthik. Ai agent pull requests on github: Frequency, structure, and merge conflict rates, 2026. URL https://arxiv.org/abs/2607.04697.

John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik R Narasimhan, and Ofir Press. SWE-agent: Agent-computer interfaces enable automated software engineering. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024a. URL https://arxiv.org/abs/2405.15793.

John Yang, Carlos E. Jimenez, Alex L. Zhang, Kilian Lieret, Joyce Yang, Xindi Wu, Ori Press, Niklas Muennighoff, Gabriel Synnaeve, Karthik R. Narasimhan, Diyi Yang, Sida I. Wang, and Ofir Press. Swe-bench multimodal: Do ai systems generalize to visual software domains?, 2024b. URL https://arxiv.org/abs/2410.03859.

John Yang, Kilian Lieret, Carlos E. Jimenez, Alexander Wettig, Kabir Khandpur, Yanzhe Zhang, Binyuan Hui, Ofir Press, Ludwig Schmidt, and Diyi Yang. Swe-smith: Scaling data for software engineering agents, 2025. URL https://arxiv.org/abs/2504.21798.

John Yang, Kilian Lieret, Jeffrey Ma, Parth Thakkar, Dmitrii Pedchenko, Sten Sootla, Emily McMilin, Pengcheng Yin, Rui Hou, Gabriel Synnaeve, Diyi Yang, and Ofir Press. Program-bench: Can language models rebuild programs from scratch?, 2026. URL https://arxiv.org/abs/2605.03546.

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

Jenny Zhang, Bingchen Zhao, Wannan Yang, Jakob Foerster, Jeff Clune, Minqi Jiang, Sam Devlin, and Tatiana Shavrina. Hyperagents, 2026a. URL https://arxiv.org/abs/2603.19461.

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

Shaoqiu Zhang, Maoquan Wang, Yuling Shi, Yuhang Wang, Xiaodong Gu, Yongqiang Yao, Tori Gong, Sheng Chen, Rao Fu, Anisha Agarwal, Spandan Grag, Gabriel Ryan, Colin Merkel, Yufan Huang, and Shengyu Fu. Fastcontext: Training efficient repository explorer for coding agents, 2026b. URL https://arxiv.org/abs/2606.14066.

Shaoqiu Zhang, Yuhang Wang, Jialiang Liang, Yuling Shi, Wenhao Zeng, Maoquan Wang, Shilin He, Ningyuan Xu, Siyu Ye, Kai Cai, and Xiaodong Gu. Swe-explore: Benchmarking how coding agents explore repositories, 2026c. URL https://arxiv.org/abs/2606.07297.

Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. Autocoderover: Autonomous program improvement, 2024. URL https://arxiv.org/abs/2404.05427.

A Limitations

CoDev-Bench is limited in scale. Its 206 pairs limit the statistical power of analyses on small subsets, such as a single language or a single combination of task types. The scale is also hard to grow: of the 3,226 candidate pairs that reach validation, about 94% are discarded, mostly because the environment cannot be built or the hidden tests do not fail before and pass after the gold patch. This is a challenge shared by SWE-Bench-style benchmarks generally (Jimenez et al., 2024).

B Dataset Construction Details

Table 4 reports the full attrition from the raw candidate pool to the final 206 pairs. The pipeline trades yield for validity: execution-level gates dominate the attrition. Building each pair's environment and requiring a fail-to-pass (F2P) signal, i.e., the hidden tests fail before the gold patch and pass after it, removes 85% of the 3,226 pairs that reach this stage. The problem-statement and test audit, a per-PR F2P check (the hidden tests must still fail when either PR's changes are left out, so that each PR is needed), and the final manual audit remove a further 57%.

Execution gates. In the environment gate, each pair is placed into a reproducible Docker environment, where the hidden tests must pass the F2P check and the developer and test patches must modify disjoint file sets; this reduces 3,226 pairs to 476. Candidates that pass the problem-statement and test audit (Appendix C) then face the per-PR F2P check, which discards any candidate whose hidden tests still fully pass when either PR's changes are left out, and a file-level provenance check confirming that the hidden test patch touches files attributable to both PRs, followed by the final manual audit.

Table 4: Seven-layer attrition of the CODEV-BENCH construction pipeline. Percentages are relative to the preceding row.

Stage	Count	Reduction
Clean repositories (post-blacklist)	4,817	–
Mined PR pairs (three merge styles)	43,170	(mining)
Pair-level filter (size, bots, revert, docs)	19,716	-54%
Window filter (time overlap, completeness)	4,200	-79%
Six-language whitelist	3,226	-23%
Environment build + F2P check	476	-85%
Problem-statement/test audit + per-PR F2P check + manual audit	206	-57%

Dataset statistics. The surviving 206 pairs have the following properties. Merge dates span July 2014 to May 2026 (median July 2023). Each pair's test patch adds or modifies a mean of 8.3 test files (median 6, max 82). Language distribution by pairs: Python 29%, Go 28%, TypeScript 17%, Java 16%, C# 6%, Rust 5%. The repositories are predominantly organization-owned; 99 of the 137 repositories contribute one pair each. PRs are classified by development activity using Conventional Commits (Conventional Commits, 2019) prefixes (e.g., `feat:`, `fix:`) and keyword heuristics, with LLM-assisted backfill for ambiguous titles (labels released with the benchmark): after feature additions (54.1% of 412 PRs) come bugfixes (21.8%), refactors (9.5%), removals (3.6%), and updates (3.2%), with the remaining 7.8% spread across less frequent prefixes such as perf and chore; the third most common pair combination, after feature+feature (34.5%) and bugfix+feature (20.4%), is feature+refactor (7.3%).

C PROBLEM-STATEMENT AUTHORING AND TEST AUDIT

Problem-statement authoring. Problem statements are written from the merged gold diff in three LLM passes. An explorer summarizes what user-visible capability each PR adds; a writer turns this summary and the public PR context (title, body, diff stats) into a three-part statement (what changes, why, reproduction steps); a granularity judge accepts the statement or returns it for rewriting. All three passes enforce a shared granularity rubric: describe the capability, never the mechanism, so the statement says what the change achieves without revealing source paths, internal helpers, or naming schemes.

Hidden-test fairness audit. An LLM judge, rechecked by human experts, examines every assertion in each pair's hidden tests that decides whether a test passes (setup and scaffolding code are skipped). Pairs whose assertions mostly check how the maintainer happened to write the code rather than what it does (exact identifiers, internal strings, snapshot bytes, call counts) are discarded, since a functionally correct but stylistically different patch could fail them; pairs whose assertions test externally observable outcomes are admitted, as are mixed suites in which behavioral assertions dominate. Every pair that survives the subsequent execution gates receives a final manual audit of its problem statements and hidden tests before entering the manifest.

Models. Both the problem-statement authoring and hidden-test labeling passes use GPT-5.5. All LLM-generated outputs, including problem statements and fairness labels, are reviewed and corrected by human experts before entering the final manifest.

D DETAILED EXPERIMENTAL RESULTS

D.1 MAIN RESULTS AND ATTRIBUTION

Full results with standard deviations. Table 5 reports the mean and standard deviation of Resolved Rate over three trials for all 28 configurations.

Trial stability. Figure 9 shows, per model (averaged across all 7 configurations), the share of pairs solved in all three trials (pass^3) and in at least one (pass@3). Roughly half of each cell's solves are stable across trials: for Opus $\times$ Mini-SWE-Agent Single Agent, 14.6% of pairs are solved in every trial while 26.7% are solved in at least one, bracketing the 20.9% mean. Even the three-trial union

Table 5: Main results with standard deviations: Resolved Rate (%, mean±sd over three trials).

	Mini-SWE-Agent			Claude Code CLI		Copilot CLI	
Model	Single Agent	Git Merge	LLM Merge	Single Agent	Sub-Agent	Single Agent	Sub-Agent
Claude Opus 4.8	20.9±0.8	20.7±1.0	21.4±1.3	20.6±1.0	21.7±0.7	21.7±1.2	**21.8**±0.8
Grok 4.5	18.6±1.5	19.7±0.6	**21.4**±1.0	21.0±1.0	19.1±0.3	18.6±1.2	18.9±0.8
GPT-5.6-Sol	15.7±1.0	16.2±1.5	**16.5**±1.5	14.6±0.5	13.8±0.3	14.6±1.3	13.9±1.0
Kimi K2.6	12.0±0.3	14.2±0.3	**14.4**±0.3	12.1±0.8	11.7±1.3	13.4±1.0	12.1±0.8

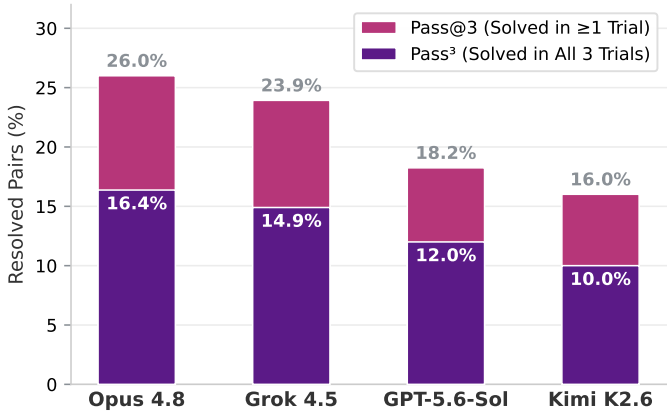


Figure 9: Trial stability per model (averaged across 7 configurations). Each bar is stacked: the lower segment is pass^3 (solved in all 3 trials) and the full height is pass@3 (solved in $\geq$1 trial). The gap reflects trial-to-trial variability.

never exceeds 27.2% in any cell. The gap between pass^3 and pass@3 is similar across models (6–10 percentage points), indicating comparable trial-to-trial variability.

Discriminative power. Under Mini-SWE-Agent Single Agent, 73.3% of task pairs are failed by all four models, 9.7% are solved by all four, and Fleiss' κ (Fleiss, 1971) across the models is 0.674, within the "substantial agreement" band (0.61–0.80) of Landis & Koch (1977): the difficulty is a stable property of the tasks rather than noise, and the spread in Table 2 separates capability tiers. 41.3% of pairs (85/206) are solved by at least one configuration on the first trial, and 43.2% (89/206) over all three trials, well above any single configuration.

Per-side evaluation checks. Per-side evaluations (§4.2.3) execute only the given side's hidden tests and must pass three validity checks: (1) *gold check*: applying the full two-sided gold patch makes the side's tests pass, proving that the side's tests are executable and judged correctly; (2) *empty-patch check*: applying no patch must fail, proving that the evaluation genuinely constrains the fix; (3) *per-side gold check*: applying that side's own gold patch alone passes the side's evaluation, either directly or after the test files belonging only to the other side are deleted (in compiled languages these files reference the other side's code and would otherwise break the build), proving that the side can be evaluated independently of the other.

Separability base rate. Applied to the whole manifest rather than to failures only, the same per-side machinery measures separability as a task property: a pair is separable when, for each side, (i) the evaluation can be restricted to that side's hidden tests, (ii) these tests pass the gold and empty-patch checks above, and (iii) they pass the per-side gold check, directly or after the deletion described there. Of the 206 pairs, 92 are separable and 114 (55%) are not. The two groups fail at different gates: 83 fail (iii)—one side's hidden tests pass only when the other PR's changes are also applied—while the other 31 fail (i) or (ii), where the hidden tests cannot be restricted to one side or the restricted set does not pass the gold and empty-patch checks. In 41 of the 83, the two PRs share no file. The 58.2% inseparable share among Grok's failures (§4.2.3) is close to this manifest-wide base rate. Table 6 crosses separability with each model's Mini-SWE-Agent Single Agent outcome: inseparable pairs make up 41.7–46.3% of every model's solved pairs and 57.1–58.2% of its failures,

and at least one of the four models solves 28 of the 92 separable pairs (30%) versus 27 of the 114 inseparable ones (24%).

Table 6: Task-level separability crossed with Mini-SWE-Agent Single Agent outcome.

Inseparable share	Grok	Opus	GPT-5.6-Sol	Kimi
among solved pairs	18/41 (43.9%)	19/41 (46.3%)	14/33 (42.4%)	10/24 (41.7%)
among failed pairs	96/165 (58.2%)	95/165 (57.6%)	100/173 (57.8%)	104/182 (57.1%)

D.2 Ruling Out Alternative Explanations

File overlap. Disjoint (N=113) and overlapping (N=93) pairs show no significant difference in Resolved Rate for any of the four models. Because the three trials of a pair are strongly correlated, we test at the pair level rather than treating the 618 runs as independent: for each pair we count the trials (0–3) in which it is solved, take the difference in the mean count between the two groups as the statistic, and compare it with 200,000 random reassignments of the overlap label across pairs. Under Single Agent, $p = 1.00$ for Opus, 0.61 for Grok, 0.50 for GPT-5.6-Sol, and 0.20 for Kimi; under LLM Merge, all $p \geq 0.40$ (Git Merge: all $p \geq 0.21$).

Skipping one PR. Regardless of outcome, 97–99% of trajectories make substantive edits on both sides with no systematic skew in per-side edit depth; runs that edit only one side account for 1.0–3.4%.

Added volume. The union of the two gold patches averages 17.1 non-test files, roughly twice the 8.9 of a single side. Grok's Resolved Rate falls with gold size (28% on pairs of at most 10 files, 17–18% at 11–40, and none of the 11 pairs beyond 40). The drop coincides with coupling rather than incapacity: across these four size bins (72, 83, 40, and 11 pairs), the share of pairs classified as inseparable rises (36%, 52%, 50%, 64%) while the share of incapacity failures does not (35%, 30%, 32%, 18%); both shares are taken over all pairs in the bin. Incapacity failures are also only slightly larger than the tasks Grok solves (median 12 versus 11 gold files, against 15 for inseparable pairs). Across the four models, failed runs edit slightly more code than correct ones (11.4 versus 9.1 files on average).

D.3 Decomposition Analysis

Timing. Mini-SWE-Agent split's effective wall-clock time is level with Single Agent (single versus multi, median: Grok 21.3 versus 22.7 minutes, GPT-5.6-Sol 11.0 versus 11.4, Kimi 15.2 versus 16.0) and 10% faster for Opus (13.5 versus 12.1); the CLI sub-agent mode is slower in 7 of the 8 comparisons, with the largest slowdown GPT-5.6-Sol $\times$ Claude Code, from 22.1 to 36.2 minutes.

Paired comparisons. Table 7 reports single-vs-multi resolved counts and McNemar exact p-values for each trial.

Split gain/loss decomposition (Grok, Mini-SWE-Agent split-and-merge). Against Single Agent, Git Merge gains 8, 12, and 10 pairs and loses 7, 7, and 9 over the three trials; LLM Merge gains 8, 13, and 12 and loses 5, 6, and 5. Inseparable pairs (Appendix D.1) account for 15 of Git Merge's 30 gains and 14 of its 23 losses, and for 18 of LLM Merge's 33 gains and 9 of its 16 losses, close to their 55% share of the benchmark. Part of Git Merge's losses come from the merge itself: in 3, 2, and 4 of them, one worker's patch no longer applies on top of the other's and is dropped.

Sub-agent dispatches. The dispatch count of §4.4.2 covers 3,459 dispatches on Claude Code and 2,897 on Copilot. Copilot dispatches carry the framework's own `agent_type` field, so exploration is counted directly; Claude Code has no role field, so review dispatches are matched by a keyword rule over the description and prompt. Copilot sub-agents rarely touch the repository at all, while on Claude Code only Grok's edit substantially (1,431 edit calls, against 24–138 for the other three).

Table 7: Paired single-vs-multi comparisons per trial (206 pairs). Each cell gives single/multi resolved counts and the McNemar exact p.

Model	Comparison	Trial 1	Trial 2	Trial 3
Opus 4.8	Mini-SWE-Agent: Single Agent vs. Git Merge	41/45 (.50)	44/42 (.79)	44/41 (.63)
Opus 4.8	Mini-SWE-Agent: Single Agent vs. LLM Merge	41/47 (.24)	44/43 (1.0)	44/42 (.79)
Opus 4.8	Claude Code: Single Agent vs. Sub-Agent	43/46 (.58)	44/45 (1.0)	40/43 (.51)
Opus 4.8	Copilot: Single Agent vs. Sub-Agent	42/44 (.79)	47/44 (.58)	45/47 (.80)
Grok 4.5	Mini-SWE-Agent: Single Agent vs. Git Merge	41/42 (1.0)	35/40 (.36)	39/40 (1.0)
Grok 4.5	Mini-SWE-Agent: Single Agent vs. LLM Merge	41/44 (.58)	35/42 (.17)	39/46 (.14)
Grok 4.5	Claude Code: Single Agent vs. Sub-Agent	44/39 (.27)	45/40 (.27)	41/39 (.73)
Grok 4.5	Copilot: Single Agent vs. Sub-Agent	41/40 (1.0)	38/37 (1.0)	36/40 (.48)
GPT-5.6-Sol	Mini-SWE-Agent: Single Agent vs. Git Merge	33/34 (1.0)	34/30 (.29)	30/36 (.11)
GPT-5.6-Sol	Mini-SWE-Agent: Single Agent vs. LLM Merge	33/34 (1.0)	34/31 (.38)	30/37 (.04)
GPT-5.6-Sol	Claude Code: Single Agent vs. Sub-Agent	30/29 (1.0)	29/28 (1.0)	31/28 (.51)
GPT-5.6-Sol	Copilot: Single Agent vs. Sub-Agent	32/27 (.27)	31/28 (.51)	27/31 (.39)
Kimi K2.6	Mini-SWE-Agent: Single Agent vs. Git Merge	24/29 (.23)	25/29 (.22)	25/30 (.27)
Kimi K2.6	Mini-SWE-Agent: Single Agent vs. LLM Merge	24/29 (.23)	25/30 (.06)	25/30 (.27)
Kimi K2.6	Claude Code: Single Agent vs. Sub-Agent	23/21 (.73)	26/25 (1.0)	26/26 (1.0)
Kimi K2.6	Copilot: Single Agent vs. Sub-Agent	27/23 (.29)	30/26 (.34)	26/26 (1.0)

E Trajectory Profiles and Case Studies

We segment each trajectory into exploration, editing, and testing stages, using the position of the first edit and the distribution of test executions as boundaries, group the statistics by attribution category, and give a representative case for each.

Single-task incapacity: verification continues to the very end, and the run still fails by a handful of tests. These runs execute the most tests of the three categories (activity rate 86.2%, median 5 test executions, the last a median of 2 steps before submission); the budget runs out with a few tests still failing. In sqlfluff/sqlfluff, Grok finishes at 259/260 after ten targeted test runs, all green, failing a single hidden test that pins the exact order of two output lines, and the per-side worker assigned to that PR fails the same assertion, confirming that the obstacle is the subtask itself rather than the concurrency.

Separable: each side is solvable on its own, and the failure appears only when one agent carries both. The workers establish per-side solvability, so this category isolates the cost of the dual-task arrangement itself. It has 4 pairs, three of them missing a single point at the finish of one side and the fourth failing on both (§4.2.3). facebook/docusaurus shows what the second task changes, because the same model solves the same subtask under two conditions. The worker, holding that side alone, implements the required git-date utility in the shared utils package that the hidden test imports from, and passes. The single agent, holding both sides, implements the same utility inside the blog plugin it is already editing, and rewrites the visible test's import to match, so its self-tests stay green while the hidden test fails. The difference between the two runs is the second task. The process signature agrees: these runs test about as often as successful ones (median 3 versus 2 executions), but 1 of the 4 pairs never obtains a complete test result before submitting, and the docusaurus run's three jest invocations all target individual test files rather than the suite. The category's relation to decomposition's actual gains is taken up in Appendix D.3.

Inseparable (together with separable pairs whose crashes are attributed to interaction): no signature in the trajectory. Although this is the largest category, it cannot be identified from the trajectory alone: its stage profile is nearly identical to that of the incapacity category, and both sides receive substantive work. The coupling is observable only in outcomes, through the attribution gates and the hidden tests; no stage of current agents' workflows verifies the interaction between the two changes. google/timesketch shows what a typical run in this category looks like. Both PRs modify sketch.py and its test file; Grok completes one side's pagination change and the other side's model refactor inside the same method, changing the repository's default page size from 40 to 100 along the way and updating the visible test to 100 as well. Its twelve test invocations therefore all succeed, two of them running the whole suite (159 tests, all green), while the hidden tests pin the repository's original default: exactly 1 of the 28 hidden tests fails, on an assertion that differs only in the limit

and size keys. The error sits inside the method both PRs rework, and the agent's own verification has been aligned with it, so the trajectory carries no signal.

Verification does not scale with task size. Pooling all runs rather than splitting them by category, verification effort also fails to grow with the task. On Mini-SWE-Agent Single Agent runs (trial 1, 206 per model), the number of files an agent edits grows with the size of the combined gold change (Spearman $\rho = 0.40$–0.65 across the four models), but the number of test executions does not ($\rho = -0.09$–0.01). The share of runs that never execute a test rises from 19–25% on pairs whose gold changes touch at most 10 files to 32–50% on pairs touching 21–40 files; overall, 29–42% of runs never execute a test, and only 1.9–7.8% run the repository's full test suite. Testing frequency alone does not separate correct from wrong runs: for Opus and Grok, correct runs skip testing about as often as wrong ones (32% versus 35% and 34% versus 27%).

Case studies. Cases from §4.3: the two forms verification takes in Mechanism 1, and the task dependency behind Mechanism 2, each with the trajectory, the patch, and the evaluation log the claim rests on.

Mechanism 1: verification does not cover the change (saleor/saleor, Grok 4.5, Mini-SWE-Agent Single Agent, trial 2)

Task. PR A extends product-variant `where` filtering; PR B threads the active channel context through the `Attribute` GraphQL types.

Run. 72 of 350 steps, $2.69 of $10, exit *Submitted*. The trajectory contains no test command: the only checks are a `py_compile` pass over the edited modules and one Django import probe, which times out and is never retried.

```
step  88   timeout 90.venv/bin/python...  django.setup()
          → [rc=124]
step  89   ``Django setup seems to hang -- possibly waiting
          on the database.''
step 104   ``Both features are implemented in a unified
          change set.  Submitting.''
```

Patch. Its version of PR B routes the top-level `attributes` query through the channel-aware context; the gold patch wraps the same queryset with `channel_slug=None`.

```
-  def resolve_attributes(self, info, *, search=None, **kwargs):
+  def resolve_attributes(self, info, *, search=None,
+                         channel=None, **kwargs):
+      qs = ChannelQsContext(qs=qs, channel_slug=channel)
```

Outcome. The hidden-test run stops at 72 passed, 1 failed:

```
AssertionError:  [{'message':  'More than one channel exists.
  Specify which channel to use.', 'path':  ['attributes']}]
```

The failing test, `test_attributes_in_collection_query[inCategory]`, is neither added nor modified by the hidden test patch: it passes at the base commit and under the gold patch. One run of the repository's existing attribute tests would have surfaced the defect. Ten of the 84 runs on this pair, from two models and all three frameworks, fail on that test with that message.

Mechanism 1, second form: the check confirms the assumption it shares with the code (slackapi/python-slack-sdk, GPT-5.6-Sol, Mini-SWE-Agent Single Agent, trial 3)

Task. PR A adds a `cursor` argument to the Audit Logs `logs` method; PR B restores 2.x-compatible handling of the `Retry-After` header on HTTP 429 responses, so callers can read it under either casing. 81 of the 84 runs on this pair solve the task.

Run. $0.20 of $10, exit *Submitted*. The agent implements PR A, then argues that PR B needs no work at all. Its evidence is one experiment it runs in the container:

```
m = Message(); m['retry-after'] = '1'
print('Retry-After' in m, 'retry-after' in m, m['Retry-After'])
    → True True 1
```

from which it concludes, in its own words, that "Feature B is already implemented in the baseline … and aiohttp's `CIMultiDictProxy` supports both casings through mapping access." Its patch touches two files, adds the `cursor` argument, and contains no change for PR B.

Patch. The client does not hand callers that `Message`; it copies the headers into a plain dict, which is case-sensitive. The gold patch is one line:

```
- resp = {"status": e.code, "headers": e.headers}
+ response_headers = dict(e.headers.items())
+ resp = {"status": e.code, "headers": response_headers}
```

Outcome. 19 passed, 1 failed:

```
test_web_client.py:90: in
    test_slack_api_rate_limiting_exception_returns_retry_after
    self.assertEqual(1,
        int(slack_api_error.response.headers["Retry-After"]))
E   KeyError: 'Retry-After'
```

The agent did test, and its test passed. It tested the object it had been reading rather than the one the library hands to callers, so the check encoded the same assumption as the decision it was meant to justify.

Mechanism 2: implicit coupling through a shared abstraction (nginx/kubernetes-ingress, PRs #1413 and #1444)

Task. PR A adds a `serverSnippets` field to the `TransportServer` resource; PR B makes a route's proxy action emit exactly one `Host` header. Both edit `virtualserver.go`, and neither problem statement mentions the other change.

PR B gives every proxied location a default header:

```
+  if !hasHostHeader {
+      headers = append(headers,
+          version2.Header{Name: "Host", Value: "$host"})
+  }
```

The dependency. PR A's own hidden tests compare the whole generated configuration structure, so they encode PR B's default. With only PR A's gold change applied, the test `TestGenerateVirtualServerConfig` reports a single field mismatch:

```
   generateVirtualServerConfig() mismatch (-want +got):
-      ProxySetHeaders: []version2.Header{{Name: "Host",
-                                          Value: "$host"}},
+      ProxySetHeaders: nil,
```

What the agents do. PR A's hidden tests carry the default `Host` header that only PR B produces, and no wording of PR A's problem statement removes that dependency, because passing PR A's tests requires PR B's behavior. What it costs an agent is that the header has to come out of the container PR A's assertions read. Of the twelve Mini-SWE-Agent Single Agent runs, Opus and Grok put it in the shared `ProxySetHeaders` list — where the repository's existing request-header machinery already lives, and where PR B's own requirement that `requestHeaders.set` *override* the default rather than add to it points as well. GPT-5.6-Sol and Kimi instead emit the header through a dedicated template field, skipping `Host` inside `generateProxySetHeaders`, so that list never carries the entry. Their implementations satisfy every behavior PR B's statement asks for, and the same decision settles whether PR A's expected configuration can be met.

Outcome. With both gold changes applied, PR A's side of the hidden tests passes 394/394; with PR A's change alone, 381 pass and 10 fail, four of them tests that PR A itself modifies. The relation is symmetric: PR B's side goes from 146/146 to 132 passed, 5 failed. No file list or merge tool reports a conflict between the two changes. This is the structure Mechanism 2 counts: 83 of the 206 pairs have at least one such side.

F Experimental Protocol and Isolation

Versions and protocol. CLI versions are pinned across all runs (Claude Code 2.1.251, Copilot CLI 1.0.74). Thinking settings are pinned per model at its default and held identical across the three frameworks: Claude Opus 4.8 uses adaptive thinking at effort high, Grok 4.5 uses effort high, GPT-5.6-Sol uses effort medium, and Kimi K2.6 runs with thinking enabled, which is its only mode. Resource limits follow standard practice in SWE-Bench-style evaluations: Mini-SWE-Agent single

Table 8: List prices used for all reported costs (USD per million tokens, July 2026). Every request is priced at the default context tier; – means no separate cache-write charge.

Model	Input	Cached input	Cache write	Output
Claude Opus 4.8	5.00	0.50	6.25	25.00
Grok 4.5	2.00	0.30	–	6.00
GPT-5.6-Sol	4.00	0.40	5.00	20.00
Kimi K2.6	0.95	0.16	–	4.00

runs are capped at 350 steps or $10 (workers at 250 steps / $5, as is the LLM Merger); the CLIs run under a uniform $30 cost cap; a run that exhausts its cap without a passing patch counts as a failure. Mini-SWE-Agent's SWE-Bench configuration defaults to 250 steps and $3 per instance,[2] which we raise for our larger tasks; Git Merge incurs no additional LLM calls, and the CLIs do not expose a step counter. All three solver architectures run in the same container and are scored by the same hidden tests; exact invocations ship with the released code.

- **Single Agent (Mini-SWE-Agent).** A shell-tool agent capped at 350 steps and $10, with a 300-second limit per command; command outputs beyond 10,000 characters are truncated.
- **Single Agent (CLIs).** Claude Code and Copilot CLI run non-interactively with permission prompts disabled, web tools removed at the tool layer, and the sub-agent model pinned to the main model. In single-agent mode, Copilot's sub-agent-orchestration tools are additionally removed.
- **Split-and-merge (mini).** Two workers, each running the single-agent setup at 250 steps / $5, independently solve one PR side each. In LLM Merge, a merger agent (250 steps / $5) sees only the two worker patches and produces the unified patch; it is forbidden to run install, build, or test commands or to modify lockfiles. In Git Merge, no model is involved: the two patches are applied in turn with `git apply`, and a patch that does not apply cleanly is dropped (3.6% of runs), so Git Merge slightly understates what a full three-way merge could recover.
- **Sub-Agent (CLIs).** The single-agent configuration with the framework's agent-dispatch tool available and a prompt directive to delegate sub-tasks to sub-agents and integrate their results.

Model versions and pricing. Average cost per pair spans $1.11–3.30 for Mini-SWE-Agent Single Agent and $2.04–15.11 across CLI configurations. All reported costs are computed by our own accounting from raw per-request token usage at the list prices of each model's provider (Anthropic, xAI, Azure OpenAI, and Fireworks AI; Table 8).

Runtime isolation. Each task runs inside a sealed Docker container with four layers of isolation. Code-hosting and search domains are blocked at the DNS level through the container's hosts file. Network egress is restricted: Mini-SWE-Agent containers run with networking disabled entirely, and an automated egress probe must fail before each session starts. The repository is stripped of history beyond the base commit: all branches, tags, remote refs, packed refs, and reflogs are deleted, and unreachable objects are garbage-collected, so post-base commits cannot be recovered from local git objects. File-mode tracking is disabled to suppress spurious diffs from the overlay filesystem.

Contamination checks. All gold patches are public merged commits, so the isolation above prevents lookup during solving but not memorization during training. We run two checks. First, under memorization, solved pairs should skew older than failed ones. For three of the four models the merge-date distributions of solved and failed pairs are statistically indistinguishable (Mann-Whitney, $p = 0.24$–0.45); for Opus the difference is marginal ($p = 0.058$), with solved pairs skewing older. Second, on the 33 pairs merged after July 2025, Resolved Rates are lower for all four models, significantly so only for Opus (3.0% versus 23.1%, Fisher $p = 0.007$); these recent pairs are also larger, but the Opus gap persists after adjusting for size (logistic $p = 0.037$). On these data we cannot distinguish memorization of older gold patches from lower familiarity with newer code.

[2] `https://github.com/SWE-agent/mini-swe-agent/blob/147121ff35ba/src/minisweagent/config/benchmarks/swebench.yaml`

G Problem Statement Example

We illustrate a concrete CoDev-Bench task using pair QuantConnect/Lean (an *overlapping* pair; 19 non-test files and 1,429 lines, against benchmark means of 17.1 files and 507 lines). The agent receives the two problem statements, one per PR, each written by the three-pass authoring pipeline (Appendix C); the upper panel reproduces them with a few sentences cut. The lower panel gives what the agent does not see: a summary of the two PRs and three of the hidden-test assertions.

Visible to the agent: problem statements (excerpted)

Feature A (PR #6213): Feature implement ExchangeInfoDownloader
What changes. Implement exchange-info downloader support for GDAX/Coinbase and integrate Bitfinex into the same exchange-info downloader/updater workflow used to refresh the symbol properties database. Generated symbol-properties rows should be based on normalized exchange metadata, be deterministically ordered, and remain compatible with the existing symbol properties CSV format. Refreshing a market must replace that market's existing rows while preserving unrelated markets and comments; [. . .]
Why. This automates keeping crypto symbol properties in sync with exchange metadata, including values such as product identifiers, quote currencies, price increments, lot sizes, and minimum order sizes. [. . .]

Feature B (PR #6222): Implement Binance US exchange
What changes. Add first-class Binance.US support as a distinct crypto exchange, selectable through the public market and brokerage APIs separately from global Binance. Binance.US crypto securities should resolve their own symbol properties and 24/7 UTC market hours, and data conversion/mapping should preserve Binance.US as its own market. The Binance.US brokerage model should use Binance.US defaults, support cash accounts only, and apply 1x leverage, while existing Binance behavior remains unchanged.
Why. Binance.US is a separate cryptocurrency exchange intended for United States residents and should not be modeled by reusing the global Binance market.

Not visible to the agent: PR summary and hidden tests

PR A (#6213)	Implements an `IExchangeInfoDownloader` interface for GDAX and Bitfinex, with HTTP-header support and deterministic CSV output for the symbol-properties database.
PR B (#6222)	Adds Binance.US as a distinct market with its own brokerage model, symbol properties, and market hours, separate from global Binance.
Overlap	Both PRs modify the symbol-properties CSV database; otherwise they touch related but distinct subsystems (exchange-info downloader vs. brokerage model).

The hidden test patch touches 9 test files (2 new, 7 modified). Three of its assertions:

```
A  Assert.IsTrue(tickers.Any());
B  Assert.AreEqual(Market.BinanceUS,
       defaultMarkets[SecurityType.Crypto]);
B  Assert.Throws<ArgumentException>(() =>
       new BinanceUSBrokerageModel(AccountType.Margin));
```

They check that the downloader returns a non-empty ticker list, that the Binance.US brokerage model defaults crypto to the Binance.US market, and that it rejects margin accounts: externally observable outcomes rather than implementation details.

H Data and Code Availability

Release. We will release the benchmark data (the 206-pair manifest with task-type, language, and file-overlap labels, frozen container images, and per-pair evaluation kits with hidden tests), the evaluation toolkit with exact agent invocations, and all agent and evaluation trajectories, including per-run cost records. The evaluation harness is available for review now at `https://anonymous.4open.science/r/codev-bench-42AZ`; the task data, trajectories, and cost records are released on publication. In the released records each run carries a boolean `resolved` and an `outcome` field; Resolved, Wrong Answer, and Crash correspond to `correct`, `wrong_answer`, and `eval_crash`.

Licenses. All 137 source repositories carry permissive open-source licenses: Apache-2.0 (57.7%), MIT (35.8%), BSD-3-Clause (4.4%), or BSD-2-Clause (2.2%). Two of these, for which GitHub's classifier reports no recognized license, were reconciled manually: unocss/unocss is standard MIT with a non-standard copyright line, and in the split-licensed weaviate/weaviate every file our pair modifies lies in BSD-3-Clause sub-trees. Container images redistribute base-commit snapshots with their original LICENSE and NOTICE files.

Privacy. All data derive from public repositories and their pull-request metadata. The released manifest and problem statements contain no usernames or other personal information; the same-author flag used during pair filtering is a boolean.