DECENTRALIZED MULTI-AGENT COORDINATION AND COMMUNICATION IN SHARED CODEBASES

Anonymous authors
Paper under double-blind review

ABSTRACT

Coding agents working in shared repositories may receive separate, independently-motivated requests from human developers. When this happens, their individual implementations may work separately but conflict when combined. To study this setting, we introduce ParallACT, a benchmark of orchestrator-free multi-agent coding tasks. ParallACT contains both complementary and contradictory tasks. In complementary tasks, each agent's task is separately useful and implementations must share an interface to be productively combined. In contradictory pairs, both agents' tasks require editing the same code in incompatible ways, and naively combining independent solutions fails to produce a useful implementation. We compare a single-agent oracle given the combined set of tasks against multi-agent settings that vary execution order, visibility, awareness of other agents, and access to a communication channel. We find that the oracle solves most tasks while parallel agents perform substantially worse. Their performance drop is notable for complementary tasks, but sharp (e.g., $62\% \rightarrow 16\%$) for contradictory tasks where merged code typically satisfies only one specification. Simply telling an agent that a peer exists does not help significantly, and giving agents access to a communication channel only recovers part of the performance when it enforces synchronization, which comes at an additional token cost. ParallACT thus identifies and isolates a core challenge that will become more common as multi-agent coding by independent principals becomes standard, and provides a controlled test of which coordination strategies help, and at what cost. Data/evaluation code: https://anonymous.4open.science/r/ParallACT/

1 INTRODUCTION

The digital world is driven by large codebases often worked on by hundreds or thousands of contributors simultaneously. Recently, coding workflows have begun to include a new kind of contributor: agents powered by large language models (LLMs), which can transform complex natural language specifications from a developer into code (Yang et al., 2024). In principle, LLM agents seem to be a good fit for large-scale development: indeed, they are increasingly being used in multi-agent settings, with large swarms of agents working in parallel towards a shared goal (Qian et al., 2025; Lin, 2026). This setting is illustrated in Fig. 1 (left), where a single principal gives a task to a central planner, who decomposes it into compatible subtasks and distributes them across agents. However, real-world multi-agent coding environments often differ from this centralized picture. In large-scale development, principals may not share goals, have conflicting specifications, and may not even be aware of each other, as illustrated in Fig. 1 (right). Naively implementing specifications in this setting can lead to code that fails to meet either specification.

One way to avoid conflicts is through communication and coordination, both explicit (e.g., sending messages to other developers, opening issues or pull requests) and implicit (e.g., inferring the likely goals of others from the environment or history, and shaping code to be compatible with those goals). Past work has studied communication between coding agents (Khatua et al., 2026) but has focused on a team setting, in which agents are aware that they are working on shared tasks with other agents. This kind of mutually-aware agent deployment has recently been validated at scale on problems such as proof search (Xin et al., 2025; OpenAI, 2026b). We instead study **what happens when principals who may not be aware of each other delegate tasks to agents in environments**

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.

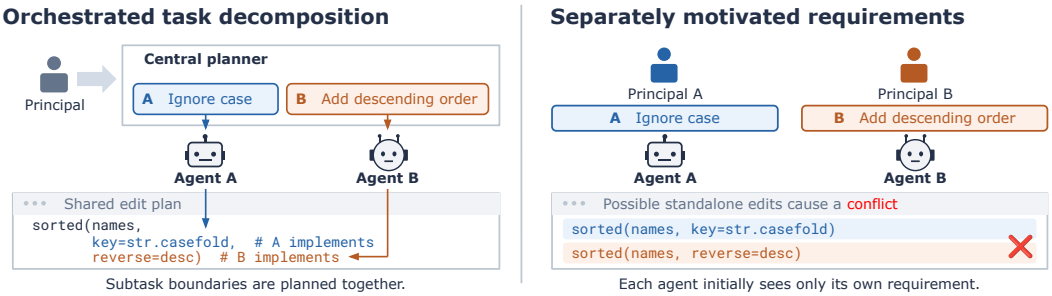


Figure 1: Two settings illustrated with the same sorting task. A central planner sees both requirements and plans the subtasks together (left). With separate principals, each agent initially sees only its own requirement, and individually valid edits may cause conflict on the same code (right).

which lack a central orchestrator and may feature conflicting or contingent specifications. This is a critical question: assuming that agents will increasingly be taking over writing code from human developers, they will have to work in large, shared codebases where many other agents are concurrently implementing changes, without any explicit pre-coordination between agents. Moreover, in addition to understanding the performance of agents under different multi-agent and coordination settings, we would like to measure the cost of different coordination approaches.

To study these questions, we introduce **ParallACT** (Parallel Agent Coordination Testbed), a diagnostic dataset of 85 challenging, self-contained multi-agent coding tasks. The goal of ParallACT is to provide a realistic and targeted testbed for understanding the collaborative abilities of LLM agents in orchestrator-free, multi-principal, multi-agent settings. Moreover, while our problem is motivated by settings with large teams, we consider pairs of agents – the minimal unit for multi-agent coordination – allowing us to isolate coordination failures in a controlled fashion. ParallACT consists of a manually-constructed gold split, with positive specifications sourced from SWE-Bench Pro (Deng et al., 2025) and manually-constructed competing specifications, and a silver split, which is automatically constructed by GPT 5.6 Sol using gold examples as seeds. In total, ParallACT contains 50 contradictory and 35 complementary instances, with each instance having been manually verified (Appendix A); simplified examples of each family are shown in Fig. 2. Contradictory instances provide competing specifications, and are filtered for examples that feature a true behavioral conflict: we ensure that reference solutions that meet each specification independently cannot be merged into a valid joint solution. Complementary instances, on the other hand, require agents to agree upon a shared interface, and are tested via integration tests; here, we filter examples to ensure that no independent solution to either agent's problem passes the tests on its own. These filters ensure that success requires deep coordination between agents.

Using our data, we test open- and closed-weight frontier models (GLM 5.3 Flash, DeepSeek V4 Flash, and Claude Opus 4.8). We first establish that the instances provided are achievable by a single developer: when both specifications are provided to a single agent, success rates are fairly high, with Claude Opus 4.8 achieving 62% success rate on contradictory and 86% on complementary instances, which increases to 98% and 100% when the agent is permitted to debug using provided tests. We then test different multi-agent topologies and communication strategies. To set a topline of what can be done with a manually-specified topology, we test our problems under sequential and re-entrant settings, where agents take turns editing the codebase (e.g., Agent A, then Agent B, potentially iterating in the re-entrant case). Under this fixed setting, we find that agents perform worse than the single-agent oracle but still succeed at least half of the cases, in both conditions. However, this setting is unrealistic; while it may be possible to specify fixed topologies for $n = 2$ agents, it quickly becomes infeasible as n increases (e.g., imagine scaling to 100 agents, where each agent must block on 99 other agents before making its edits).

We then test agents across multiple parallel settings, where agents work on their specifications simultaneously. Here, we study several ad-hoc communication strategies, optionally providing agents with access to either a single shared file to write updates to, or a Redis cache (Appendix L.1) in which they can send messages. While each agent generally succeeds in implementing solutions that pass their own respective tests, they categorically fail to coordinate on a joint solution, with two mutually-unaware Claude Opus 4.8 agents achieving an average joint pass rate of only 16% on

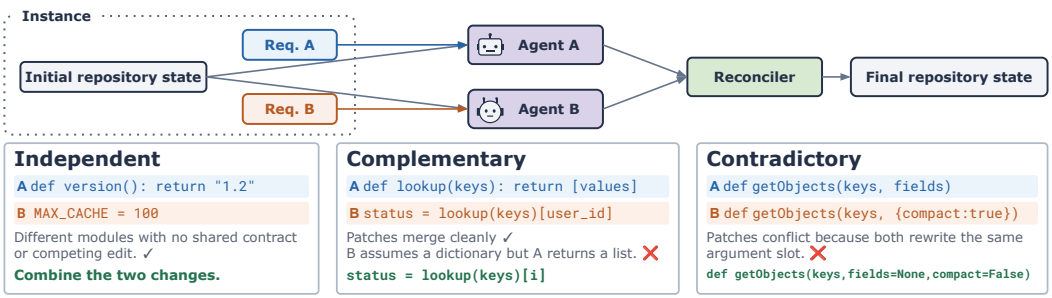


Figure 2: An illustration of an instance with three examples: (a) independent edits and a clean merge; (b) complementary contract mismatch despite a clean merge; (c) contradictory edits competing for the same slot. Green text shows possible reconciliations.

contradictory and 57% on complementary tasks. While these drops can sometimes be mitigated by adding communication and making agents aware of each other, performance remains substantially lower than in the single-agent condition. Moreover, we find that communication increases costs.

Building on these results, we analyze the failure modes of different topologies, finding that single-agent failures can be traced to capability shortcomings while multi-agent settings are bottlenecked by coordination failures. Moreover, we study the communication channel in the parallel setting, finding that agents often succeed in terms of *what to communicate* – communicating about the right topics (e.g., the shared artifact) – but often fail in terms of *how to communicate and act on communication*, e.g., failing to read messages on time or at all, or agreeing to a design but not following through.

2 Problem Statement

We study a scenario common in large-scale software engineering: *multiple programmers pursuing separate tasks on a shared codebase*, each delegating work to their own coding agent. Each agent is responsible for its own programmer's request and may be *unaware* of other agents or their requirements, even when their changes affect the same codebase or must interoperate through a shared interface. To isolate the coordination challenges in this broader scenario, we focus on the smallest nontrivial case: two agents with independently motivated and jointly satisfiable requirements. This setting captures conflicts and dependencies that can also arise within larger teams. We examine how peer awareness, communication, and execution schedules affect joint correctness and token cost.

An *instance* consists of an *initial repository state* and two separate *requirements* in natural language, each requesting a change to that state. Each requirement is assigned to a separate coding agent, and is disclosed only to its agent by default. The requirements are created separately without a central planner who sees both and decomposes the work. Through code changes and, in some cases, communication, the agents must produce a single *final repository state* that satisfies **both** requests.

Broadly, requirements can interact in three different ways within the initial repository: (a) **Independent** requirements admit separate implementations that can be correctly combined with only surface-level reconciliation; (b) **Complementary** requirements specify individually motivated tasks that must agree on a shared interface or contract to satisfy an integration test, but either implementation alone fails; (c) **Contradictory** requirements specify tasks that require implementing or modifying the same code region in ways that would fail when mechanically combined, but that can be reconciled to satisfy both tasks. We focus on the latter two families, as these require non-trivial coordination between agents. Each requirement is associated with a set of *acceptance conditions* designed as executable tests that assess different aspects of the desired behavior. We evaluate the final repository state against the acceptance conditions of both requirements, on the assumption that the final repository should satisfy both users' goals. *Joint success* requires all conditions to be satisfied, while the fraction of satisfied conditions provides a more lenient measure of partial completion.

A *coordination strategy* specifies four aspects of execution: (a) The **execution schedule** determines when each agent acts and whether their actions overlap in time; (b) The **observation model** determines what each agent can see of the repository and the other's work; (c) The **communication model** determines whether and how agents can exchange information beyond changes to the code;

(d) The **combination rule** determines how separately produced changes become one final state, including any permitted reconciliation or repair. These choices determine both the information available to the agents during implementation and the opportunities to resolve incompatible decisions.

We evaluate coordination strategies by **joint success rate**, **completion rate**, and relative effective **token cost**. A single agent given both requirements provides an *oracle* for implementation with access to the full task specification – we consider this to be an oracle because it deviates from our motivating setting, in which multiple principals implement changes in parallel. In this setting, a single agent corresponding to a single principal has access to all task-relevant information at once. This measures to what degree the models we are testing are capable of executing on the requested specifications, setting a topline for their abilities. Relative to this ceiling, we ask how joint success changes when agents work concurrently without awareness of one another. We then examine which components of a coordination strategy recover success and at what cost, whether effective communication is constrained by the available channel or by how agents use it, and how much a combination rule can repair after separate implementation.

3 Benchmark Construction and Experimental Settings

This section describes how we construct ParallACT and the coordination scenarios tested.

3.1 Benchmark Construction

Data sourcing. We build ParallACT starting from SWE-Bench Pro. For any given instance in our dataset, we take requirement A from the original SWE-Bench Pro task and introduce a new requirement B, which is a separately-motivated task that a different developer could be tasked with, which merely happens to touch the same code. In contradictory instances, tasks A and B collide on shared code, while in complementary tasks they must agree on a shared interface.

Testing and filtering. Each instance starts from an initial repository state and includes a problem statement for each of task A and B, a per-side reference solution for each (a correct implementation of that one requirement), and a set of acceptance tests. We store a reference solution as a *patch*: a set of edits that, when applied to the original repository, implements that task's requirement. For contradictory instances, we additionally add a combined test set, which runs both tasks' acceptance tests together, and a *reconciliation solution*: a single implementation that simultaneously satisfies both requirements. The reconciliation solution is distinct from the two per-side reference solutions since each of those solve their respective tasks; because A's and B's solutions collide on the shared code, stacking them does not give a working joint solution. For complementary instances, we define an integration test that passes only when both sides agree on the interface they share. Outside the oracle, communication oracle, and gold-test conditions (defined in Section 3.2), problem statements refer only to their own requirements and do not mention the other task, its files, or any of the tests.

We filter instances for soundness according to the following procedure. Starting from the repository at its original commit as defined in SWE-Bench Pro, we apply: task A's patch by itself, task B's patch by itself, both patches applied one after the other, and both patches combined through a Git merge. After each of these applications, we run the tests that apply to that case – a single side's own acceptance tests, the combined test set, or the integration test – and filter based on the following rules, which ensure that the two requirements interact as intended and that no naive combination of the per-side solutions already produces a valid joint solution. A contradictory instance is sound when: (a) The unmodified initial repository fails both sides' tests. (b) Each agent's reference solution applied alone passes its respective tests. (c) Applying concatenated patches fails the combined test set. (d) Applying both via a git merge also fails the combined test set. (e) The reconciliation passes the combined set. A complementary instance is sound when: (a) Each agent's reference solution applied alone fails the integration test. (b) Applying both together passes the integration test.

Seed set. We first hand-build a gold set of *seed* instances (the manually constructed instances counted in Appendix A) each developed from one SWE-Bench Pro task, taken as task A. Task B, its reference solution, its tests, and the family-specific artifacts (the combined test set and reconciliation, or the integration test) are written by the authors, ensuring they pass the filtering conditions. Each seed instance was further checked manually after passing the filters.

Data generation. We scale the benchmark beyond the seeds by generating further instances with GPT 5.6 Sol in Codex (Appendix L.2), again keeping requirement A as a SWE-Bench Pro task and authoring requirement B as an independent counterpart. Diversity is enforced by selecting candidate tasks via a deterministic, stratified sweep over repository metadata (first programming language, then repository). We use a smaller model (Appendix L.3) to route candidates to the complementary or contradictory group. We then prompt GPT 5.6 Sol to produce task B requirements and artifacts, using tasks from the seed set as in-context examples. After authoring, each instance is checked against the soundness conditions for its family, and any failure triggers an automatic correction round in which GPT 5.6 Sol revises the instance and it is re-checked; we allow up to two such rounds, after which an instance that still fails any condition is discarded. We give the full generation procedure in Appendix B and give the full breakdown by repository and family in Appendix A.

3.2 Experimental Settings

We evaluate each instance under *coordination scenarios* that vary three things: the *schedule* (when each agent acts and whether they overlap), the *observation model* (what each agent sees of the repository and the other's work), and the *communication model* available beyond code. A *turn* is one agent invocation on a workspace. We run 19 contradictory and 18 complementary scenarios; Section 3.3 describes how changes are combined and scored, and Fig. 3 (Appendix) shows the four groups compared: single-agent oracle, serial hand-offs, parallel execution, as well as gold-test access. The exact prompt text for every scenario is shown in Appendix M.

Single-agent references. *Oracle* gives one agent both requirements and asks for the whole solution, an upper bound requiring no coordination. *Orchestrator* also gives one agent both requirements but asks it to delegate each to a separate subagent and do no implementation itself.

Serial (hand-offs). Two agents edit the same workspace in turn, each starting from the previous result. *Sequential* runs two turns in one order (A then B, or B then A); *reentrant* runs four alternating turns (ABAB or BABA). In both, we additionally vary the observation model: whether the acting agent sees only the current file state (*repo*) or also sees the earlier agent's file edit history (*hist*). Note that this strategy scales poorly as the number of agents increases.

Parallel. Agents work simultaneously from the same base, and their changes are kept separate until evaluation. They are either *unaware* that a peer exists, *aware* of a peer but without a communication channel, or given a two-way message channel framed as free-form *messaging* or as explicit *synchronization* points. The framings differ only in how agents are prompted to use the channel: under *messaging* they may send and read at any time and in any form, whereas under *synchronization* they treat the channel as explicit checkpoints, announcing each intended change and reconciling it with the peer before continuing. We run both framings over two mediums. In the first, agents share a single append-only file: to send, an agent appends its message; to receive, it re-reads the file for new entries. In the second, messages go through Redis, where each agent has its own inbox: sending pushes a message into the peer's inbox, and receiving reads and clears the agent's own. The send/receive interface is identical in both cases. Finally, a *communication oracle* extends parallel messaging by giving each agent its partner's requirement verbatim, with an instruction to communicate and solve both problems together. This isolates whether using the channel – rather than missing information about the peer's task – is the bottleneck.

Gold-test access. Two further oracle-style scenarios establish performance upper bounds by letting agents debug against the gold tests paired with each instance. *Oracle + gold tests* gives the single-agent oracle the gold tests, and *parallel + gold tests* gives the parallel-unaware agents their respective tests. The parallel variant covers only contradictory instances, since complementary instances have only a shared integration test, which cannot evaluate one agent's implementation without the other's.

3.3 Evaluation Methods and Metrics

Merge policies. Parallel scenarios require merging: each agent produces an independent edit from the same starting point, and the two must be combined into a single codebase before grading. We combine them in three ways of increasing sophistication, following Khatua et al. (2026). *Concatenation* applies both edits together; *Git merge* merges them as two branches off the base and treats a merge conflict as a failure; and *reconciliation* goes further, resolving any conflict with an off-the-

shelf merge-resolver model from Khatua et al. (2026) (Appendix L.4) before grading. If any yields code that passes, the scenario is considered passed. Thus, the metric reflects the best a downstream merge step could recover without further agent work. Appendix F shows a per-policy breakdown.

Metrics. We report two primary axes: joint success (pass rate) and cost. For contradictory instances we also report partial completion: the fraction of instances where agent A's tests pass, where agent B's pass, where both pass, and where either passes (Appendix E). This can identify cases where each side succeeds alone but the agents fail together. Because true agent cost differs across models and platforms, as well as across token type (input vs. output, cache hits vs. misses, etc.), to obtain a single token cost number, we report cost based on the following formula: $\text{input} + \text{cache-write} + 0.1 \cdot \text{cache-read} + 5 \cdot \text{output}$. The weights approximate Claude Opus 4.8's per-token rates, with output counted at $5\times$ and cache reads at $0.1\times$ fresh input, and cache-write counted the same as fresh input as a proxy (Appendix L.5). We report cost relative to the oracle rather than as absolute dollars, since prices are billed differently across providers and drift over time, and since the three models use different tokenizers. For the cross-model comparison (Table 2; full results in Appendix D) we drop the cache-write term since Claude Code does not report cache-write tokens for the open-weight models; this keeps the effective-token count comparable across models (see Appendix L.5)). The underlying raw per-scenario token counts are given in Appendix H.

Harness. We use Claude Code (Appendix L.6) as our agent harness. Claude Opus 4.8 runs natively through Claude Code with reasoning effort set to `high` and its full 1M-token context window; the open-weight models are driven through the same harness by routing Claude Code to OpenRouter's Anthropic-compatible endpoint (Appendix L.7), where the reasoning-effort control is not exposed, so we instead fix their thinking budget at $32{,}000$ tokens.

Models and evaluation subset. We run Claude Opus 4.8 on the full benchmark , and add two open-weight models, DeepSeek V4 Flash and GLM 5.3 Flash (Appendices L.8 and L.9), on a 40-instance subset (20 contradictory and 20 complementary). The subset is chosen deterministically, based on the assumption that the open models will perform worse than Claude Opus 4.8. Thus, we filter for cases which Claude Opus 4.8 solved in the oracle setting (so the instance is solvable by a strong model); this avoids unnecessarily running models on instances they will not solve. We then stratify instances, such that all repositories, families, and languages are represented in roughly the same proportions as in the full data. Every selected instance is still run under the full set of scenarios, and each cross-model comparison is computed on these identical instances (Table 2 and Appendix D).

4 Results & Analysis

We report results for Claude Opus 4.8 on the full dataset and then report performance across DeepSeek V4 Flash and GLM 5.3 Flash on the subset. We compare scenarios on the same instances using two-sided exact McNemar tests for joint success and paired Wilcoxon signed-rank tests for effective token costs. Full significance test results appear in Appendix I.

4.1 Results

Performance drops dramatically in parallel settings. We first compare the single-agent oracle with parallel-unaware, where each agent receives only its own requirement and works independently. **Joint pass rates drop significantly** from 62% to 16% on contradictory instances ($p < 0.001$) and from 86% to 57% ($p = 0.02$) on complementary instances (Table 1; Appendix I). This drop is also reflected when agents have access to gold tests: here, the oracle agent achieves a near-perfect pass rate of 98% on contradictory examples. In contrast, the parallel-unaware agent with gold tests achieves only 38% success ($p < 0.001$). Thus, **access to gold tests alone does not recover the oracle's joint success** in the parallel setting, indicating that agents fail to coordinate on shared problems even when they can debug against gold tests.

Sequential baselines recover performance but do not scale. To assess sequential access to the evolving codebase, we compare parallel-unaware with a single A→B hand-off, in this case without access to edit history. Joint success rises significantly from 16% to 54% on contradictory instances ($p < 0.001$). It also increases from 57% to 77% on complementary instances, though this is non-significant ($p = 0.065$). The mean effective token cost remains similar: 1.17 to 1.18 times the oracle for contradictory instances, and 1.14 to 1.12 times for complementary instances (Table 1).

Table 1: Pass rate (%) and relative cost of Claude Opus 4.8 on the full benchmark, for a subset of the conditions. *Cost* is mean effective tokens relative to the family's `oracle` run (Section 3.3). Among the deployable strategies the top two pass rates per family are in **bold**. Darker red shading indicates worse, darker blue shading indicates better. Full per-split results and variants are in Table 4.

Upper Bounds					*Deployable*				
	Contradictory		Complementary			Contradictory		Complementary	
Scenario	Pass	Cost	Pass	Cost	Scenario	Pass	Cost	Pass	Cost
Oracle	62	1.00	86	1.00	Seq. A→B	**54**	1.18	**77**	1.12
Orchestrator	58	1.58	77	1.18	Reentr. ABAB	**50**	1.88	**80**	1.49
Comm-Oracle	56	2.74	83	1.31	Parallel Unaware	16	1.17	57	1.14
Oracle+Gold	98	0.74	100	0.54	Parallel Aware	26	1.14	60	1.21
Parallel+Gold	38	0.77	n/a	n/a	Parallel Message	20	1.42	63	1.43
					Parallel Sync	22	1.63	71	1.43

Therefore, **a single hand-off recovers much of the observed performance gap at comparable token cost**. We further extend this in the reentrant scenario (ABAB) that allows both agents a second turn. This has no significant effect on the pass rate, achieving 50% on contradictory (4% drop) and 80% (3% gain), while raising costs significantly to 1.88 and 1.49 times the oracle. In Appendix D we show similar results for sequential and reentrant baselines when the orders are reversed, as well as when we add history. Based on these results, we conclude that **reentrance costs significantly more and provides no discernible performance gains.** Moreover, *enforcing sequential or reentrant turn-taking is impractical with more agents*; if hundreds or thousands of developers independently deploy agents on a shared codebase – as might be expected in large organizations and teams using monorepos – sequential turn-taking becomes completely infeasible.

Peer awareness and access to communication marginally improve performance but fail to close the gap. We compare parallel agents with peer awareness, messaging, and synchronization. Peer awareness slightly raises joint success from 16% to 26% on contradictory instances and from 57% to 60% on complementary instances, but neither gain is significant. Messaging and synchronization also provide limited improvements, reaching 20%/63% (contradictory/complementary) and 22%/71%, respectively (Table 1). We also find that the communication model matters: Redis Sync reaches 32%/77%, the highest success rates among parallel variants with ($p = 0.04$ for contradictory); however, this comes at a greater effective token cost of 2.23/1.39 times the single-agent oracle (Appendix D), and still trails the single-agent oracle by 30 and 9 percentage points. Note that all four messaging and synchronization variants use more effective tokens than parallel-unaware. The communication oracle reaches 56%/83%, exceeding Redis Sync by 24 and 6 percentage points (Appendix D). The fact that a gap remains to the single-agent oracle performance even when the agent is exactly instructed on what to communicate about and how suggests that providing a communication channel is insufficient, particularly for contradictory tasks. We conclude that **communication improves joint success at a higher token cost, but leaves a substantial gap to the communication oracle on contradictory tasks.** We further analyze the contents of the communication channel in Section 4.2.

Post-hoc reconciliation fails to address coordination failures. To separate joint success from the choice of combination rule, Appendix F evaluates concatenated diffs, Git merging, and a model-based reconciler taken from Khatua et al. (2026) on the integration of code changes from parallel agents. For parallel-unaware, contradictory success is 10% (concatenation), 14% (merging), and 16% (model-based), while complementary success is 51%, 54%, and 54%. These effects differ when agents are given a communication channel: in the Redis sync setting, we see a larger improvement from reconciliation that raises contradictory success from 22% to 32%, and complementary success from 69% to 77%. Moreover, across all four non-oracle parallel conditions shown in Table 1 (and indeed, across all six such conditions in the full results in Appendix D), the reconciler remains below the sequential A→B baseline's 54% contradictory success. None of these differences is significant (Appendix F). In other words, **there is no evidence that coordination failures observed in the parallel setting are resolved by changing the combination rule alone.**

Parallel-unaware trends hold across models, but others differ. In Appendix D, we evaluate a matched subset of 40 instances on three models: Claude Opus 4.8, GLM 5.3 Flash, and DeepSeek V4 Flash. In general, DeepSeek V4 Flash and GLM 5.3 Flash perform worse than Claude Opus 4.8

Table 2: Cross-model comparison on the subset where every cell is computed on the identical instances. Columns, reference-vs-deployable grouping, and bolding follow Table 1; *Cost* is relative to each model's own `oracle`, excluding cache-write tokens for cross-model comparability (Section 3.3). The full scenario set is in Appendix D.

	Contradictory						Complementary					
	Opus 4.8		DeepSeek-v4		GLM-5.3		Opus 4.8		DeepSeek-v4		GLM-5.3	
Scenario	Pass	Cost	Pass	Cost	Pass	Cost	Pass	Cost	Pass	Cost	Pass	Cost
Oracle	95	1.00	60	1.00	90	1.00	100	1.00	55	1.00	65	1.00
Orchestrator	95	1.42	30	1.60	70	0.86	85	1.02	30	0.54	55	0.37
Comm-Oracle	90	2.66	35	3.67	70	1.22	90	1.21	40	1.36	75	0.37
Oracle + Gold Tests	100	0.61	65	0.81	95	0.88	100	0.53	15	0.36	85	0.78
Parallel + Gold Tests	55	0.68	15	1.08	35	0.92	n/a	n/a	n/a	n/a	n/a	n/a
Sequential A→B	**70**	1.02	**30**	1.83	**85**	0.83	**85**	1.09	**30**	0.64	**90**	0.26
Reentrant ABAB	**80**	1.62	**45**	3.58	**70**	2.07	**90**	1.33	**35**	1.47	**85**	1.69
Parallel Unaware	20	1.13	15	1.83	30	1.92	55	1.00	**30**	0.88	65	0.29
Parallel Aware	45	1.04	0	1.68	30	0.88	65	1.06	20	0.91	55	0.38
Parallel Message	35	1.27	15	2.34	40	3.29	75	1.24	25	1.34	70	0.44
Parallel Sync	35	1.49	10	3.42	40	1.55	80	1.36	20	1.77	80	0.36

but are comparable in some cases. Here, we find that all three models show a substantial oracle-to-parallel performance gap, especially on contradictory tasks. Relative to parallel-unaware, awareness helps Claude Opus 4.8, is neutral for GLM 5.3 Flash, and hurts DeepSeek V4 Flash; communication helps Claude Opus 4.8 and GLM 5.3 Flash but rarely DeepSeek V4 Flash. Communication often raises token cost for both open models and leaves a performance gap to the communication oracle on contradictory tasks. Reentrance costs more and has a mixed effect on pass rates. Thus, **all models have a parallel performance gap and higher communication cost, but the benefits of awareness, communication, and turn-taking vary across models.**

4.2 Analysis

Error analysis and failure modes. We manually open-coded every failing Opus run over the oracle and parallel scenarios, grounding each code in the produced patches, the gold solutions, and the message logs, and then regrouped the codes into a small set of classes. The isolated per-agent test results let us label each contradictory failure objectively as either a *competence* failure (an agent's patch fails its own tests) or an *integration* failure (both patches pass alone, but none of their combinations passes). We find that **single-agent failures are mostly competence failures:** 92% of oracle failures involve the oracle producing an implementation that diverges from the test requirements (e.g., logic placed in the wrong function or layer, wrong API signatures, return values, or configurations). On the other hand, **parallel failures are mostly integration failures:** across parallel conditions, in 58% of contradictory failures each agent's half passes its own tests but no combination of the two patches passes the joint tests, and 97% of those produce a git merge conflict. The dominant complementary mechanism is different: 53% of complementary failures merge cleanly but fail because the two halves disagree on a name, signature, or data shape. Finally, **communication does not prevent conflicting edits and sometimes causes them:** even when the agents agree on a correct joint design in the channel, the run still fails due to changing text style (variable names, quote style, import order), both agents delegating merging to each other, or avoiding the conflicting code entirely. This finding extends those of Khatua et al. (2026) that agents deviate from commitments even after effective communication. In our setting the agents often attempt to keep their commitments and still fail.

Communication analysis. We further analyze the message logs and transcripts of all valid Claude Opus 4.8 channel runs, track when each message is read relative to the recipient's first edit, and label the conversation content with an outcome-blind LLM judge (Claude Sonnet 5) validated against human labels; full details are in Appendix J. In summary, agents know *what* to communicate but largely fail in terms of *who* should act and *how* to act on information in a timely fashion. First, **agents send early but read late.** Communication is two-way in at least 94% of runs and most agents send messages before editing, yet about $1/3$ of messages are never read, and only 21–23% are read before the recipient's first edit, because agents check for messages mainly at the start and end of

a turn. This gels with the findings of Khatua et al. (2026). However, unlike Khatua et al. (2026), who found that agents often polluted the channel "with vague and inaccurate messages", we find that the agents we tested **discuss the right thing**. 88% of contradictory and 76% of complementary conversations address the annotated shared artifact, and 59% and 74% respectively reach a concrete agreement on names, signatures, or edit ownership. Finally, **knowing the partner's task leads to subtask ownership:** in the communication oracle, 85% of contradictory conversations assign an owner to the contested code, conflicts fall from 65% to 15% of runs, and runs that assign an owner pass at a higher rate of 61% versus 29%. Interestingly, in 15% of conversations, one agent implements *both* requirements while the other withdraws. Note that this is an unscalable strategy that defeats the purpose of multi-agent collaboration, since with n agents it is completely infeasible for one agent to do all the work on a company scale.

5 Related Work

Coordination in Classical Multi-agent Systems. Dependencies, conflicts, and communication costs are longstanding concerns in multi-agent systems, which has studied the selective exchange of local plans (Durfee & Lesser, 1988), beneficial and conflicting plan interactions (Von Martial, 1992), and trade-offs between communication and miscoordination costs (Tambe, 1997). We study similar problems in a complex, real-world setting: *decentralized software development*, where individual agents must implement natural language requirements in a shared codebase.

Coding Agents and Multi-agent Development. Coding agent harnesses (e.g., Claude Code (Anthropic, 2026), Codex (OpenAI, 2026a), SWE-Agent (Yang et al., 2024), OpenHands (Wang et al., 2025), etc.) equip LLMs with tools to navigate repositories, edit files, and execute code. We study how multiple agents operating in such harnesses coordinate on shared code. Multi-LLM systems such as MetaGPT (Hong et al., 2024), ChatDev (Qian et al., 2024), MAGIS (Tao et al., 2024), and AgileCoder (Nguyen et al., 2025) organize development work through predefined roles or structured dialogue, coordinating around a single shared development request. CAID (Geng & Neubig, 2026) uses a central manager to plan dependent subtasks and runs isolated workers asynchronously. Our setting fundamentally differs: in ParallACT, multiple agents receive *separate* assignments without any central planner. This distinction is key: in centralized, decomposition-based methods coordination can be partly addressed through the decomposition itself (Hong et al., 2024; Geng & Neubig, 2026). In contrast, we study a setting where requirements are created *separately* by multiple principals, with independence enforced during dataset construction (Appendix B). Agents must discover and resolve dependencies or conflicts that are not apparent from either requirement.

Benchmarks for Agent Coordination. The Collaboration Gap (Davidson et al., 2025) studies collaborative maze solving and shows that strong solo performance does not guarantee successful teamwork. CooperBench (Khatua et al., 2026) is closest to our setting: agents operate and communicate to implement separate features on a shared repository, potentially coordinating via messages, and are scored based on a merged result. However, CooperBench focuses on a collaborative setting in which agents implementing jointly compatible features are explicitly framed as teammates, are aware of each other, and are told to communicate. We are instead motivated by a large-scale coding scenario in which agents implementing independently-motivated requests may be mutually unaware, and focus on independent tasks that conflict at a behavioral level.

6 Conclusion

As agents take on work delegated by many programmers who share a codebase but not a plan, they will increasingly face coordination challenges. We introduced ParallACT to study this decentralized multi-agent coordination setting in its most targeted form: two agents with independent requirements that must agree on a shared interface (complementary) or reconcile incompatible edits (contradictory), scored on whether their combined code satisfies both. We find that much of the difficulty lies in coordination: across models, a single agent with both requirements succeeds more often than parallel agents. Peer awareness and access to communication channels only yield marginal gains. Sequential topologies or explicit ownership assignment through communication recover much of the performance gap, but are not feasible in large-scale development settings. Finally, agents often discuss the right topics, but fail in terms of how to communicate and how to act on communication.

AI USE STATEMENT

As described in Section 3 we use LLMs to augment our manual seed set and select examples for potential annotation. We used LLMs to automate our analysis of the communication channel in Section 4.2. We also used coding agents to assist in implementing and refining experiments, cleaning and reformatting data, creating visualizations, and formatting tables. We have not used AI tools to help develop theoretical models or conceptual frameworks, formulate mathematical claims, propose or refine hypotheses, design research methodology or experiments, implement methods, assist with translation, or interpret results. The other required categories do not apply. We used LLMs to proofread and polish writing. We have reviewed all AI-assisted work. We take responsibility for the final content of this work, including text, claims or artifacts produced with the aid of generative AI.

REPRODUCIBILITY STATEMENT

We release our data and code, the link given in Section 1. We report all our dataset generation details in Appendix B, and provide prompts in the appendix as well as the repository.

REFERENCES

Anthropic. Claude Code. `https://code.claude.com/docs/en/overview`, 2026. Accessed September 25, 2026.

Mert Cemri, Melissa Z Pan, Shuyi Yang, Lakshya A Agrawal, Bhavya Chopra, Rishabh Tiwari, Kurt Keutzer, Aditya Parameswaran, Dan Klein, Kannan Ramchandran, Matei Zaharia, Joseph E. Gonzalez, and Ion Stoica. Why do multi-agent LLM systems fail? In *The Thirty-ninth Annual Conference on Neural Information Processing Systems Datasets and Benchmarks Track*, 2026. URL `https://openreview.net/forum?id=fAjbYBmonr`.

Tim R Davidson, Adam Fourney, Saleema Amershi, Robert West, Eric Horvitz, and Ece Kamar. The collaboration gap. *arXiv preprint arXiv:2511.02687*, 2025.

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.

Edmund H Durfee and Victor R Lesser. Using partial global plans to coordinate distributed problem solvers. In *Readings in distributed artificial intelligence*, pp. 285–293. Elsevier, 1988.

Jiayi Geng and Graham Neubig. Effective strategies for asynchronous software engineering agents. *arXiv preprint arXiv:2603.21489*, 2026.

Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Steven Yau, Zijuan Lin, Liyang Zhou, et al. Metagpt: Meta programming for a multi-agent collaborative framework. In *International Conference on Learning Representations*, volume 2024, pp. 23247–23275, 2024.

Arpandeep Khatua, Hao Zhu, Peter Tran, Arya Prabhudesai, Frederic Sadrieh, Johann K Lieberwirth, Xinkai Yu, Yicheng Fu, Michael J Ryan, Jiaxin Pei, et al. Cooperbench: Why coding agents cannot be your teammates yet. *arXiv preprint arXiv:2601.13295*, 2026.

Yubin Kim, Ken Gu, Chanwoo Park, Chunjong Park, Samuel Schmidgall, A Ali Heydari, Yao Yan, Zhihan Zhang, Yuchen Zhuang, Yun Liu, et al. Capable language models can outgrow the benefits of collaboration. *Nature Machine Intelligence*, 8(7):1157–1172, 2026.

Wilson Lin. Scaling long-running autonomous coding. Cursor research blog, January 2026. URL `https://cursor.com/blog/scaling-agents`. January 14, 2026.

Yuzhen Mao and Azalia Mirhoseini. Decentralized multi-agent systems with shared context. *arXiv preprint arXiv:2606.10662*, 2026.

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. In *2025 IEEE/ACM Second International Conference on AI Foundation Models and Software Engineering (Forge)*, pp. 156–167, 2025. doi: 10.1109/Forge66646.2025.00026.

OpenAI. openai/codex. https://github.com/openai/codex, 2026a. URL https://github.com/openai/codex.

OpenAI. On the Navier–Stokes millennium prize problem. https://openai.com/index/navier-stokes-solution/, 2026b. Published September 8, 2026.

Chen Qian, Wei Liu, Hongzhang Liu, Nuo Chen, Yufan Dang, Jiahao Li, Cheng Yang, Weize Chen, Yusheng Su, Xin Cong, et al. Chatdev: Communicative agents for software development. In *Proceedings of the 62nd annual meeting of the association for computational linguistics (volume 1: Long papers)*, pp. 15174–15186, 2024.

Chen Qian, Zihao Xie, YiFei Wang, Wei Liu, Kunlun Zhu, Hanchen Xia, Yufan Dang, Zhuoyun Du, Weize Chen, Cheng Yang, et al. Scaling large language model-based multi-agent collaboration. In *International Conference on Learning Representations*, volume 2025, pp. 41488–41505, 2025.

Xinxing Ren, Qianbo Zang, Ziyan Wang, Caelum Forder, Suman Deb, Peter Carroll, and Zekun Guo. Agentradio: Passive awareness for long-horizon multi-agent collaboration. *arXiv preprint arXiv:2607.28430*, 2026.

Milind Tambe. Towards flexible teamwork. *Journal of artificial intelligence research*, 7:83–124, 1997.

Wei Tao, Yucheng Zhou, Yanlin Wang, Wenqiang Zhang, Hongyu Zhang, and Yu Cheng. Magis: Llm-based multi-agent framework for github issue resolution. *Advances in Neural Information Processing Systems*, 37:51963–51993, 2024.

Frank Von Martial. *Coordinating plans of autonomous agents*. Springer, 1992.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, et al. Openhands: An open platform for ai software developers as generalist agents. In *International Conference on Learning Representations*, volume 2025, pp. 65882–65919, 2025.

Ran Xin, Zeyu Zheng, Yanchen Nie, Kun Yuan, and Xia Xiao. Scaling up multi-turn off-policy rl and multi-agent tree search for llm step-provers. *arXiv preprint arXiv:2509.06493*, 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, 2024.

A Dataset Composition

Table 3 shows the composition of ParallACT instance broken down by the repository they are from and counts of the seeds and generated instances in Contradictory and Complementary categories.

Table 3: Dataset composition. Instance counts per repository and family, split into manually constructed *Seed* instances and *Generated* instances produced by GPT 5.6 Sol using seed examples for guidance. Instances are produced by adding complementary or contradictory tasks to SWE-Bench Pro instances; thus, they reflect coding problems from real-world repositories. ParallACT comprises of 85 instances (50 contradictory + 35 complementary). Examples are drawn from 10 open-source repositories across four language types.

		Contradictory			Complementary		
Repo	Language	# Seed	# Generated	Total	# Seed	# Generated	Total
flipt	Go	2	2	4	1	1	2
navidrome	Go	0	6	6	3	4	7
teleport	Go	0	1	1	1	1	2
vuls	Go	1	4	5	1	3	4
ansible	Python	2	7	9	2	1	3
openlibrary	Python	2	1	3	2	2	4
qutebrowser	Python	2	4	6	1	1	2
NodeBB	JS/TS	2	6	8	2	3	5
element-web	JS/TS	1	2	3	2	3	5
webclients	JS/TS	0	5	5	0	1	1
Total		12	38	50	15	20	35

B Dataset Generation Details

This section of the appendix gives details on how candidate tasks are selected, how the scout, author, and auditor stages operate.

Candidate selection and diversity by construction. Diversity is enforced before any model inspects the code, so the candidate set is spread by construction. Starting from the pool of SWE-Bench Pro tasks, each task is labeled from row metadata alone: its language, its repository, and the subsystem it edits (the dominant changed source directory in the task's gold patch). A deterministic, stratified selector then fills a fixed number of candidates by balancing first on language, then on repository (with a floor so every repository, including ones untouched by the seeds, is represented), then round-robin over distinct subsystems, capped at two candidates per (repository, subsystem) pair. The selection uses a fixed random seed and a stable sort so it is reproducible, and it prefers subsystems not already used by an existing seed to maximize cross-batch novelty.

Suggestion stage. On this pre-spread candidate set, a scout model (Qwen3.7 Flash via OpenRouter) explores each task's repository and proposes which family the task best supports and how strong the conflict or interface signal is, under the independent-task-pair framing (requirement A the original standalone task, requirement B a separately motivated counterpart). The scout's judgment is an explicit, non-authoritative suggestion: it is used only to rank and route candidates into the authoring queue.

Authoring. For each selected candidate, GPT 5.6 Sol (in Codex) authors requirement B together with its reference solution, its tests, and the family-specific artifacts (the combined test set and reconciliation for contradictory instances; the integration test for complementary ones). The author is shown two in-context examples drawn from the seeds of the same family, chosen by a uniform similarity score (same repository, then same language, then same code area) so the harness, test-id, and formatting conventions match the repository being authored. The authored instance is then validated (every diff must apply and every schema field must be present) before verification.

Verification and correction. Verification runs inside the original SWE-Bench Pro execution environment: each source task ships a prebuilt per-task Docker image, and we build and test every instance in that same image (inherited unchanged from the source task) so the runtime matches the

upstream benchmark rather than a reconstructed one. Each authored instance is verified against the soundness conditions of Section 3 by applying every relevant combination of changes in this container and running the instance's tests. An instance that fails any check is handed the failing check's diagnostic and re-authored, up to two correction rounds; instances that never become sound are discarded.

Independence and blind audit. Soundness alone does not guarantee that the two requirements read as independent tickets rather than two halves of one decomposed task. A separate auditing pass with GPT 5.6 Sol therefore checks each sound instance against the independence criterion and the blind rule (no problem statement may name the other requirement, its agent, its files, or its tests), and instances judged to be decomposition or to leak are rejected. Every instance that passes the model audit is additionally reviewed by hand.

Cross-set diversity audit. Diversity is a property of the set, not of any single instance, so in addition to diversity-by-construction at selection we periodically run a cross-set audit over the finalized instances of each family. It labels each instance by conflict archetype or capability type and by code layer, computes the resulting distributions, and flags over-concentration (a repeated (repository, subsystem) pair, a repository or an archetype that dominates the set) together with the buckets to avoid next. This audit is advisory and changes no finalized instance.

The prompt for each of these stages can be found in our code repository under `prompts/` folder

C AGENT TOPOLOGIES

Fig. 3 illustrates the topologies and coordination strategies we test. Oracle agents optionally debug against gold tests. The sequential setting is a special case of the reentrant setting, with only one iteration. In the reentrant setting, we optionally provide agents with access to the edit history. The intuition behind this is as follows: for a reentrant configuration, e.g., ABAB, if agent A is taking its second turn, and it sees that the changes it made in its first turn have been reverted by agent B, it may be less likely to simply revert the changes back, being able to detect from the history that another agent is editing the same files. On the other hand, without history, A may be more likely to simply overwrite B's edits. In the parallel setting, agents have access to a communication channel, whose implementation varies.

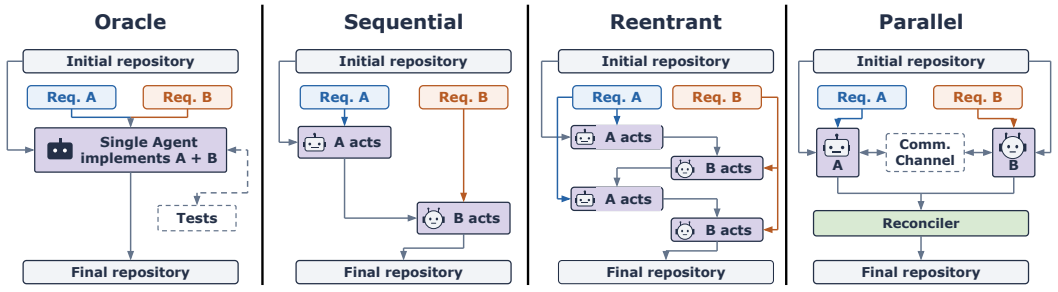


Figure 3: Coordination strategies: single-agent oracle with optional test access, sequential hand-off, reentrant, and parallel execution with optional communication and reconciliation.

D RESULTS FOR ALL SCENARIOS

Table 4 shows the performance of the agents with Claude Opus 4.8 for all the scenarios tested.

Table 5 provides a comprehensive cross-model comparison across the models tested.

Table 4: Pass rate (%) and relative cost of Claude Opus 4.8 across all 19 scenarios, on the full benchmark. *Pass* is the fraction of instances whose combined patch passes the scored test group (contradictory: union of both sides' tests; complementary: the integration test). *Cost* is mean effective tokens (Section 3.3) relative to that family's `oracle` run. Per pass column, the highest and second-highest values are in **bold** (ties shared).

	Contradictory				Complementary			
	Pass (%)				Pass (%)			
Scenario	Seeds	Gen.	All	Cost	Seeds	Gen.	All	Cost
Reference								
Oracle	67	61	62	1.00	87	85	86	1.00
Orchestrator	58	58	58	1.58	73	80	77	1.18
Comm-Oracle	50	58	56	2.74	93	75	83	1.31
Oracle + Gold Tests	100	97	98	0.74	100	100	100	0.54
Parallel + Gold Tests	17	45	38	0.77	n/a	n/a	n/a	n/a
Sequential								
A→B (Repo)	50	**55**	**54**	1.18	80	75	77	1.12
A→B (Hist)	50	50	50	1.25	80	75	77	1.15
B→A (Repo)	**58**	47	50	1.27	67	75	71	1.11
B→A (Hist)	**58**	42	46	1.26	53	**80**	69	1.11
Reentrant								
ABAB (Repo)	42	**53**	50	1.88	**87**	75	80	1.49
ABAB (Hist)	**58**	42	46	2.05	**93**	**80**	**86**	1.51
BABA (Repo)	**67**	50	**54**	1.92	**87**	**85**	**86**	1.51
BABA (Hist)	**58**	50	**52**	2.01	80	**85**	**83**	1.57
Parallel								
Unaware	0	21	16	1.17	53	60	57	1.14
Aware	0	34	26	1.14	60	60	60	1.21
Message	8	24	20	1.42	67	60	63	1.43
Sync	0	29	22	1.63	67	75	71	1.43
Message (Redis)	8	37	30	1.61	67	70	69	1.32
Sync (Redis)	8	39	32	2.23	80	75	77	1.39

Table 5: Cross-model comparison on the subset, where every cell is computed on the identical instances. *Pass* is pass rate (%); the second column per model (*Cost*) is mean effective tokens (Section 3.3) relative to that same model's `oracle` run. The *Reference* block grants privileged information and marks upper bounds. Among the deployable strategies, the highest and second-highest pass rate per column is in **bold** (ties shared). For cross-model comparability the effective-token count excludes cache-write tokens; a compact version appears in Table 2.

	Contradictory						Complementary					
	Opus 4.8		DeepSeek-v4		GLM-5.3		Opus 4.8		DeepSeek-v4		GLM-5.3	
Scenario	Pass	Cost	Pass	Cost	Pass	Cost	Pass	Cost	Pass	Cost	Pass	Cost
Reference												
Oracle	95	1.00	60	1.00	90	1.00	100	1.00	55	1.00	65	1.00
Orchestrator	95	1.42	30	1.60	70	0.86	85	1.02	30	0.54	55	0.37
Comm-Oracle	90	2.66	35	3.67	70	1.22	90	1.21	40	1.36	75	0.37
Oracle + Gold Tests	100	0.61	65	0.81	95	0.88	100	0.53	15	0.36	85	0.78
Parallel + Gold Tests	55	0.68	15	1.08	35	0.92	n/a	n/a	n/a	n/a	n/a	n/a
Sequential												
A→B (repo)	70	1.02	30	1.83	**85**	0.83	85	1.09	30	0.64	**90**	0.26
A→B (hist)	**75**	1.17	35	2.06	**80**	0.88	85	1.06	**40**	1.21	**85**	1.08
B→A (repo)	**75**	1.00	20	1.10	70	0.84	75	1.02	20	1.07	70	0.44
B→A (hist)	70	1.16	15	1.00	70	0.90	80	1.09	**40**	1.22	75	0.72
Reentrant												
ABAB (repo)	**80**	1.62	**45**	3.58	70	2.07	90	1.33	35	1.47	**85**	1.69
ABAB (hist)	70	1.87	**40**	2.92	75	1.60	**95**	1.41	**50**	1.56	80	0.58
BABA (repo)	70	1.76	**45**	3.33	75	1.56	90	1.33	**50**	1.97	70	0.87
BABA (hist)	**75**	1.74	**45**	3.01	**85**	2.62	**100**	1.39	**40**	1.74	80	0.99
Parallel												
Unaware	20	1.13	15	1.83	30	1.92	55	1.00	30	0.88	65	0.29
Aware	45	1.04	0	1.68	30	0.88	65	1.06	20	0.91	55	0.38
Message	35	1.27	15	2.34	40	3.29	75	1.24	25	1.34	70	0.44
Sync	35	1.49	10	3.42	40	1.55	80	1.36	20	1.77	80	0.36
Message (Redis)	50	1.50	5	1.87	25	0.86	80	1.17	35	1.09	75	0.37
Sync (Redis)	55	2.24	15	2.32	35	1.23	85	1.35	25	1.47	75	0.97

E PERFORMANCE BY AGENT

Table 6, Table 7 and Table 8 show the per-agent pass rate across all the scenarios tested for Claude Opus 4.8, DeepSeek V4 Flash and GLM 5.3 Flash respectively.

Table 6: Per-agent pass rates (%) for Claude Opus 4.8 on the full dataset (contradictory tasks). *A* and *B* are the fractions where agent A's and agent B's patch passes its own tests in isolation, and *Either* is at least one of them. *Both* is the combined patch passing the scored test group (the joint success rate of Table 4), a different, merged artifact. Because A/B/Either measure isolated patches while *Both* measures the merged patch, *Both* may exceed *B* and *Either* need not equal $A + B -$ Both. Per column, the highest and second-highest values are in **bold** (ties shared).

	Contradictory			
Scenario	A	B	Both	Either
Oracle	78	76	**62**	88
Orchestrator	68	76	58	86
Oracle + Gold Tests	**98**	**92**	**98**	**98**
Parallel + Gold Tests	**98**	**98**	38	**98**
Sequential A→B (Repo)	74	74	54	92
Sequential A→B (Hist)	64	80	50	92
Sequential B→A (Repo)	76	66	50	92
Sequential B→A (Hist)	**80**	62	46	92
Reentrant ABAB (Repo)	72	74	50	94
Reentrant ABAB (Hist)	58	76	46	88
Reentrant BABA (Repo)	76	74	54	92
Reentrant BABA (Hist)	78	66	52	90
Parallel Unaware	68	**92**	16	**98**
Parallel Aware	78	**92**	26	**96**
Parallel Message	72	**92**	20	94
Parallel Sync	70	**92**	22	**96**
Parallel Message (Redis)	74	88	30	**96**
Parallel Sync (Redis)	70	84	32	92
Parallel Comm-Oracle	56	70	56	86

Table 7: Per-agent pass rates (%) for DeepSeek V4 Flash on the 40-instance subset. Columns are defined as in Table 6. For contradictory, *A*/*B* are the fractions where agent A's / agent B's own tests pass, *Both* is the combined patch passing, and *Either* is at least one side passing. Per column, the highest and second-highest values are in **bold** (ties shared).

	Contradictory			
Scenario	A	B	Both	Either
Oracle	65	65	**60**	65
Orchestrator	30	40	30	40
Oracle + Gold Tests	65	65	**65**	65
Parallel + Gold Tests	60	50	15	75
Sequential A→B (Repo)	40	**75**	30	85
Sequential A→B (Hist)	55	60	35	75
Sequential B→A (Repo)	35	40	20	60
Sequential B→A (Hist)	35	35	15	55
Reentrant ABAB (Repo)	**70**	70	45	90
Reentrant ABAB (Hist)	60	65	40	85
Reentrant BABA (Repo)	60	**85**	45	**95**
Reentrant BABA (Hist)	**75**	70	45	**100**
Parallel Unaware	50	55	15	80
Parallel Aware	55	40	0	85
Parallel Message	**70**	45	15	85
Parallel Sync	45	60	10	75
Parallel Message (Redis)	50	50	5	75
Parallel Sync (Redis)	30	70	15	80
Parallel Comm-Oracle	40	60	35	75

Table 8: Per-agent pass rates (%) for GLM 5.3 Flash on the 40-instance subset. Columns are defined as in Table 6. For contradictory, *A*/*B* are the fractions where agent A's / agent B's own tests pass, *Both* is the combined patch passing, and *Either* is at least one side passing. Per column, the highest and second-highest values are in **bold** (ties shared).

	Contradictory			
Scenario	A	B	Both	Either
Oracle	85	**95**	**90**	**95**
Orchestrator	80	**95**	70	**100**
Oracle + Gold Tests	**95**	**95**	**95**	**95**
Parallel + Gold Tests	**95**	**95**	35	**95**
Sequential A→B (Repo)	85	**90**	85	90
Sequential A→B (Hist)	75	**90**	80	90
Sequential B→A (Repo)	**90**	75	70	90
Sequential B→A (Hist)	**90**	80	70	**100**
Reentrant ABAB (Repo)	80	**90**	70	**95**
Reentrant ABAB (Hist)	80	**95**	75	**95**
Reentrant BABA (Repo)	80	85	75	90
Reentrant BABA (Hist)	85	**95**	85	**95**
Parallel Unaware	**90**	**95**	30	**95**
Parallel Aware	85	**95**	30	**95**
Parallel Message	75	**95**	40	**95**
Parallel Sync	**90**	**90**	40	**95**
Parallel Message (Redis)	75	85	25	**95**
Parallel Sync (Redis)	75	**95**	35	**95**
Parallel Comm-Oracle	70	85	70	90

F MERGE POLICY

Table 9 , Table 10 and Table 11 break down parallel pass rates by merge policy for Claude Opus 4.8, DeepSeek V4 Flash and GLM 5.3 Flash.

Table 12 reports paired significance tests between merge policies for Claude Opus 4.8 on the parallel scenarios.

Table 9: Pass rate (%) of Claude Opus 4.8 on the parallel scenarios, broken out by merge policy: *concat* (concatenate the two agents' diffs), *git-merge*, *reconciler* (a post-hoc merger model), and *any* (best of the three, i.e). Per column, the highest and second-highest values are in **bold** (ties shared).

	Contradictory				Complementary			
Scenario	concat	git-merge	reconciler	any	concat	git-merge	reconciler	any
Parallel Unaware	10	14	16	16	51	54	54	57
Parallel Aware	16	18	26	26	46	49	57	60
Parallel Message	16	14	20	20	51	57	63	63
Parallel Sync	16	16	22	22	60	**69**	71	71
Parallel Message (Redis)	16	20	30	30	60	57	63	69
Parallel Sync (Redis)	**22**	**22**	**32**	**32**	**63**	**69**	**77**	**77**
Parallel Comm-Oracle	**38**	**50**	**56**	**56**	**71**	**80**	**80**	**83**

Table 10: Pass rate (%) of DeepSeek V4 Flash on the parallel scenarios (40-instance subset), broken out by merge policy: *concat*, *git-merge*, *reconciler*, and *any* (best of the three). Per column, the highest and second-highest values are in **bold** (ties shared).

	Contradictory				Complementary			
Scenario	concat	git-merge	reconciler	any	concat	git-merge	reconciler	any
Parallel Unaware	**15**	**10**	**10**	**15**	**30**	15	15	30
Parallel Aware	0	0	0	0	15	0	5	20
Parallel Message	5	**5**	**15**	**15**	15	**20**	20	25
Parallel Sync	5	**10**	**10**	10	20	15	15	20
Parallel Message (Redis)	0	0	5	5	**30**	**20**	**25**	**35**
Parallel Sync (Redis)	5	**5**	**15**	**15**	20	5	10	25
Parallel Comm-Oracle	**25**	**10**	**15**	**35**	**25**	**30**	**35**	**40**

Table 11: Pass rate (%) of GLM 5.3 Flash on the parallel scenarios (40-instance subset), broken out by merge policy: *concat*, *git-merge*, *reconciler*, and *any* (best of the three). Per column, the highest and second-highest values are in **bold** (ties shared).

	Contradictory				Complementary			
Scenario	concat	git-merge	reconciler	any	concat	git-merge	reconciler	any
Parallel Unaware	10	15	30	30	45	50	65	65
Parallel Aware	10	10	30	30	50	55	55	55
Parallel Message	**15**	10	35	**40**	50	60	70	70
Parallel Sync	10	**20**	**40**	**40**	**65**	**70**	**80**	**80**
Parallel Message (Redis)	10	10	25	25	60	**65**	**75**	**75**
Parallel Sync (Redis)	**15**	15	35	35	**65**	**65**	**75**	**75**
Parallel Comm-Oracle	**25**	**50**	**65**	**70**	**70**	**70**	**75**	**75**

Table 12: Paired comparison of merge policies for Claude Opus 4.8 on the parallel scenarios. *Passed* gives the number of instances each policy passes; *added* gives the instances gained or lost by git-merge over concatenation and by the reconciler over git-merge, with p from a binomial test on those instances. *: $p < 0.05$.

	Instances passed			git-merge − concat		reconciler − git-merge	
Scenario	concat	git-merge	reconciler	added	p	added	p
Contradictory ($n = 50$)							
Parallel Unaware	5	7	8	+2	0.500	+1	1.000
Parallel Aware	8	9	13	+1	1.000	+4	0.125
Parallel Message	8	7	10	+0/−1	1.000	+3	0.250
Parallel Sync	8	8	11	+0	1.000	+3	0.250
Parallel Message (Redis)	8	10	15	+2	0.500	+5	0.062
Parallel Sync (Redis)	11	11	16	+0	1.000	+5	0.062
Communication Oracle	19	25	28	+6	0.031*	+3	0.250
Complementary ($n = 35$)							
Parallel Unaware	18	19	19	+2/−1	1.000	+0	1.000
Parallel Aware	16	17	20	+2/−1	1.000	+3	0.250
Parallel Message	18	20	22	+2	0.500	+2	0.500
Parallel Sync	21	24	25	+3	0.250	+1	1.000
Parallel Message (Redis)	21	20	22	+1/−2	1.000	+2	0.500
Parallel Sync (Redis)	22	24	27	+2	0.500	+3	0.250
Communication Oracle	25	28	28	+3	0.250	+1/−1	1.000

G PERFORMANCE BY REPOSITORY

Table 13, Table 14 and Table 15 report per-repository oracle and mean pass rates for Claude Opus 4.8, DeepSeek V4 Flash and GLM 5.3 Flash, broken down by repository, language, and family.

Table 13: Per-repository performance of Claude Opus 4.8 on the full benchmark. n is the number of instances; *Oracle* is the single-agent oracle pass rate (%), the per-repo ceiling; *Mean* is the mean pass rate (%) across all scenarios for that repository and family.

		Contradictory			Complementary		
Repo	Lang	n	Oracle	Mean	n	Oracle	Mean
flipt	Go	4	75	41	2	100	83
navidrome	Go	6	50	39	7	100	87
teleport	Go	1	100	37	2	100	83
vuls	Go	5	60	41	4	75	65
ansible	Python	9	67	54	3	100	85
openlibrary	Python	3	67	60	4	50	72
qutebrowser	Python	6	67	48	2	100	81
NodeBB	JS/TS	8	75	48	5	100	89
element-web	JS/TS	3	33	39	5	80	61
webclients	JS/TS	5	40	33	1	0	6
All		50	62	45	35	86	76

Table 14: Per-repository performance of DeepSeek V4 Flash on the 40-instance subset. n is the number of instances; *Oracle* is the single-agent oracle pass rate (%), the per-repo ceiling; *Mean* is the mean pass rate (%) across all scenarios for that repository and family.

		Contradictory			Complementary		
Repo	Lang	n	Oracle	Mean	n	Oracle	Mean
flipt	Go	2	50	24	1	0	11
navidrome	Go	2	50	16	4	50	43
teleport	Go	1	0	11	2	50	11
vuls	Go	2	100	26	2	0	31
ansible	Python	3	100	26	2	100	28
openlibrary	Python	2	50	39	2	0	33
qutebrowser	Python	2	0	11	2	50	31
NodeBB	JS/TS	3	67	42	3	100	54
element-web	JS/TS	1	0	53	2	100	28
webclients	JS/TS	2	100	34	0	n/a	n/a
All		20	60	28	20	55	33

Table 15: Per-repository performance of GLM 5.3 Flash on the 40-instance subset. n is the number of instances; *Oracle* is the single-agent oracle pass rate (%), the per-repo ceiling; *Mean* is the mean pass rate (%) across all scenarios for that repository and family.

		Contradictory			Complementary		
Repo	Lang	n	Oracle	Mean	n	Oracle	Mean
flipt	Go	2	100	50	1	100	78
navidrome	Go	2	100	61	4	75	78
teleport	Go	1	100	16	2	50	47
vuls	Go	2	100	53	2	0	44
ansible	Python	3	100	74	2	100	89
openlibrary	Python	2	100	87	2	50	86
qutebrowser	Python	2	50	50	2	50	53
NodeBB	JS/TS	3	100	67	3	100	94
element-web	JS/TS	1	100	74	2	50	86
webclients	JS/TS	2	50	61	0	n/a	n/a
All		20	90	62	20	65	74

H ABSOLUTE TOKEN NUMBERS

Table 16, Table 17 and Table 18 report the mean per-run token counts (input, cache-read, output, and effective) for Claude Opus 4.8, DeepSeek V4 Flash and GLM 5.3 Flash across all scenarios.

Table 16: Mean per-run token counts (thousands) for Claude Opus 4.8 on the full benchmark (Section 3.3, Appendix L.5).

	Contradictory					Complementary				
Scenario	Input	Cache Read	Output	Cache Write	Effective	Input	Cache Read	Output	Cache Write	Effective
Reference										
Oracle	1.4	854.8	14.6	37.7	197.6	0.1	2525.4	26.7	72.4	458.4
Orchestrator	1.7	980.2	25.6	84.1	312.0	0.1	2400.5	35.6	123.5	541.7
Comm-Oracle	1.2	3008.6	32.4	77.6	541.8	0.1	3471.9	33.0	86.1	598.5
Oracle + Gold Tests	1.6	666.2	9.9	29.2	147.0	1.5	1223.5	15.8	42.6	245.7
Parallel + Gold Tests	2.7	642.2	9.7	36.5	152.2	n/a	n/a	n/a	n/a	n/a
Sequential										
A→B (Repo)	2.4	969.0	16.6	51.1	233.3	1.5	2744.7	28.9	91.5	511.8
A→B (Hist)	2.4	1027.7	17.7	52.3	246.1	0.8	2918.2	29.0	88.1	525.5
B→A (Repo)	2.4	1124.3	17.4	49.0	250.8	0.2	2816.8	29.1	81.9	509.4
B→A (Hist)	2.6	1116.4	17.2	48.7	249.0	0.5	2923.1	27.8	76.5	508.0
Reentrant										
ABAB (Repo)	4.8	1506.4	26.6	83.6	372.0	0.2	3587.6	39.1	127.1	681.7
ABAB (Hist)	4.9	1656.6	29.1	89.1	405.3	4.5	3784.1	38.6	118.3	694.3
BABA (Repo)	4.8	1591.9	26.8	82.2	380.4	2.2	3675.2	38.9	127.9	691.9
BABA (Hist)	6.2	1658.8	28.1	84.1	396.6	1.0	3919.0	40.8	121.9	718.5
Parallel										
Unaware	2.3	966.2	16.4	50.5	231.6	0.3	2803.3	30.4	88.5	521.0
Aware	3.2	939.2	15.9	48.7	225.1	0.3	3073.8	31.4	89.2	553.8
Message	2.5	1297.0	19.3	52.2	280.8	1.6	3866.3	34.9	92.0	654.9
Sync	2.7	1434.1	23.2	60.8	322.7	0.3	3813.3	35.6	94.1	653.5
Message (Redis)	2.5	1481.8	22.3	55.9	318.0	1.5	3501.0	32.5	89.2	603.3
Sync (Redis)	2.7	2029.1	32.4	72.4	440.1	0.9	3601.9	36.8	93.0	637.8

Table 17: Mean per-run token counts (thousands) for DeepSeek V4 Flash on the 40-instance subset (Section 3.3, Appendix L.5).

	Contradictory				Complementary			
Scenario	Input	Cache Read	Output	Effective	Input	Cache Read	Output	Effective
Reference								
Oracle	174.6	1560.8	21.8	439.6	478.1	3949.7	34.7	1046.7
Orchestrator	399.2	1727.5	26.4	703.7	166.7	2085.1	37.5	562.6
Comm-Oracle	1046.6	3703.3	39.4	1613.8	710.3	4859.6	44.8	1420.4
Oracle + Gold Tests	220.9	710.9	13.2	358.0	152.4	1560.5	13.0	373.4
Parallel + Gold Tests	209.3	1318.0	26.3	472.7	n/a	n/a	n/a	n/a
Sequential								
A→B (Repo)	445.3	2069.7	30.8	806.4	214.6	2841.6	34.6	671.6
A→B (Hist)	582.7	1733.7	30.3	907.7	759.9	2835.1	44.3	1264.8
B→A (Repo)	242.3	1438.6	19.2	482.0	536.3	4073.4	34.8	1117.7
B→A (Hist)	216.2	1166.4	21.0	438.0	801.4	2923.0	37.2	1279.7
Reentrant								
ABAB (Repo)	999.4	3341.9	47.8	1572.5	829.7	4308.8	54.8	1534.4
ABAB (Hist)	629.2	3620.3	58.1	1281.6	635.3	6391.0	72.5	1636.8
BABA (Repo)	832.7	3292.3	60.7	1465.4	1036.3	6547.3	74.1	2061.3
BABA (Hist)	727.9	3802.5	43.3	1324.7	930.5	5734.9	63.0	1819.2
Parallel								
Unaware	429.6	2069.5	33.7	805.2	379.6	3625.8	35.9	921.4
Aware	508.6	1069.1	24.6	738.5	576.9	2289.4	28.5	948.3
Message	585.9	2554.9	37.6	1029.5	828.7	3310.3	49.6	1407.7
Sync	1091.6	2373.6	35.1	1504.4	1123.6	4617.8	54.1	1855.7
Message (Redis)	487.3	1953.6	27.8	821.7	545.8	3613.2	46.3	1138.4
Sync (Redis)	517.2	2858.4	43.0	1018.1	990.4	3142.2	45.8	1533.5

Table 18: Mean per-run token counts (thousands) for GLM 5.3 Flash on the 40-instance subset (Section 3.3, Appendix L.5).

	Contradictory				Complementary			
Scenario	Input	Cache Read	Output	Effective	Input	Cache Read	Output	Effective
Reference								
Oracle	249.5	4012.9	46.9	885.5	503.5	16435.9	42.2	2358.1
Orchestrator	310.6	1914.7	52.1	762.5	308.3	3599.7	41.5	875.6
Comm-Oracle	286.4	4741.9	63.2	1076.6	311.2	3948.4	31.6	863.8
Oracle + Gold Tests	416.6	1987.7	33.1	781.0	417.9	12935.0	27.9	1851.0
Parallel + Gold Tests	404.0	2348.0	35.8	817.7	n/a	n/a	n/a	n/a
Sequential								
A→B (Repo)	266.3	2909.0	36.3	738.7	238.8	2499.5	23.3	605.3
A→B (Hist)	312.4	2757.6	38.2	779.0	687.0	16891.6	34.3	2547.4
B→A (Repo)	270.7	2787.8	39.7	748.2	371.1	4611.6	39.3	1028.9
B→A (Hist)	251.0	3387.3	41.9	799.4	605.3	9511.9	30.6	1709.4
Reentrant								
ABAB (Repo)	613.0	7898.4	85.2	1828.9	1006.8	27474.1	46.0	3984.0
ABAB (Hist)	590.7	4759.5	69.4	1413.5	624.2	5293.1	41.5	1361.1
BABA (Repo)	490.1	5278.2	71.9	1377.2	657.5	11312.0	51.0	2043.9
BABA (Hist)	821.3	10721.0	85.2	2319.4	725.2	13260.5	57.1	2336.8
Parallel								
Unaware	452.5	10268.9	44.6	1702.3	317.1	2601.2	22.9	691.6
Aware	249.0	3333.9	39.0	777.3	327.6	3970.0	33.7	892.9
Message	730.3	18506.9	66.3	2912.3	410.0	4732.1	31.6	1041.1
Sync	549.5	4765.4	69.6	1373.8	348.6	3446.1	30.3	844.7
Message (Redis)	266.2	2909.7	40.1	757.4	390.8	3453.2	26.7	869.5
Sync (Redis)	397.1	4049.9	57.1	1087.6	674.4	14121.6	40.9	2291.1

I PAIRED SIGNIFICANCE TESTS ON SCENARIOS

Table 19 reports an exact McNemar test on joint success and a paired Wilcoxon signed-rank test on effective token cost, both computed over the same instances for Claude Opus 4.8 and for every scenario comparison cited in Section 4.

Table 19: Paired significance tests for Claude Opus 4.8 between scenarios on the same instances (contradictory $n = 50$, complementary $n = 35$). *Pass* gives each scenario's joint success (%); p is an exact McNemar test on the instances that only one scenario passes. *Cost* is the ratio of mean effective tokens A/B, with the p-value of a Wilcoxon signed-rank test paired by instance. *: $p < 0.05$; **: $p < 0.01$.

	Pass (%)			Cost	
Comparison (A vs. B)	A	B	p	A/B	p
Contradictory					
single agent (oracle) vs parallel unaware	62	16	<0.001**	0.85	0.005**
single agent (oracle) + gold vs parallel + gold	98	38	<0.001**	0.97	0.002**
sequential A→B (repo) vs parallel unaware	54	16	<0.001**	1.01	0.818
reentrant ABAB (repo) vs sequential A→B (repo)	50	54	0.727	1.59	<0.001**
sequential B→A vs A→B (repo)	50	54	0.754	1.08	0.553
sequential A→B hist vs repo	50	54	0.727	1.06	0.163
orchestrator vs single agent (oracle)	58	62	0.688	1.58	<0.001**
parallel aware vs unaware	26	16	0.180	0.97	0.484
parallel message vs aware	20	26	0.453	1.25	<0.001**
parallel sync vs aware	22	26	0.688	1.43	<0.001**
parallel sync vs message	22	20	1.000	1.15	<0.001**
parallel message redis vs message (file)	30	20	0.180	1.13	0.006**
parallel sync redis vs sync (file)	32	22	0.125	1.36	<0.001**
parallel sync redis vs unaware	32	16	0.039*	1.90	<0.001**
comm. oracle vs parallel sync redis	56	32	0.002**	1.23	0.001**
comm. oracle vs sequential A→B (repo)	56	54	1.000	2.32	<0.001**
single agent (oracle) vs comm. oracle	62	56	0.508	0.36	<0.001**
single agent (oracle) vs sequential A→B (repo)	62	54	0.344	0.85	0.002**
Complementary					
single agent (oracle) vs parallel unaware	86	57	0.021*	0.88	0.017*
sequential A→B (repo) vs parallel unaware	77	57	0.065	0.98	0.777
reentrant ABAB (repo) vs sequential A→B (repo)	80	77	1.000	1.33	<0.001**
sequential B→A vs A→B (repo)	71	77	0.774	1.00	0.827
sequential A→B hist vs repo	77	77	1.000	1.03	0.104
orchestrator vs single agent (oracle)	77	86	0.375	1.18	0.023*
parallel aware vs unaware	60	57	1.000	1.06	0.245
parallel message vs aware	63	60	1.000	1.18	0.006**
parallel sync vs aware	71	60	0.344	1.18	0.001**
parallel sync vs message	71	63	0.453	1.00	0.326
parallel message redis vs message (file)	69	63	0.754	0.92	0.037*
parallel sync redis vs sync (file)	77	71	0.625	0.98	0.441
parallel sync redis vs unaware	77	57	0.065	1.22	0.003**
comm. oracle vs parallel sync redis	83	77	0.727	0.94	0.179
comm. oracle vs sequential A→B (repo)	83	77	0.727	1.17	0.004**
single agent (oracle) vs comm. oracle	86	83	1.000	0.77	<0.001**
single agent (oracle) vs sequential A→B (repo)	86	77	0.375	0.90	0.107

J Details on Communication analysis

We analyze message logs and agent transcripts from all valid Claude Opus 4.8 runs in the communication scenarios and communication oracle over both the file and Redis transports, track when messages are read relative to the recipient's first edit, and label each conversation's content. The content is labeled by an outcome-blind LLM judge (Claude Sonnet 5) validated on a stratified sample of 30 conversations labeled independently by an author. The judge agrees with the human labels on 90% of conversations with Cohen's $\kappa = 0.67$ that is substantial.

Agents often send early but read late. Communication is two-way in 94% of runs. Agents send their first message before editing in 79% of contradictory and 88% of complementary turns. However, only 67%/68% of the messages are ever read, 21%/23% are read before the recipient's first edit, and 29%/27% arrive after the recipient has already finished. This is the same unresponsiveness that CooperBench (Khatua et al., 2026) reports, where direct questions go unanswered. Agents check for incoming messages mainly at the start and end of a turn and can cause *mistimed communication*, and we find the main cause for coordination failure is ill-timing rather than unwillingness to reply.

Agents discuss the shared artifact. The validated LLM judge finds that the shared artifact is discussed in 88% of contradictory and 76% of complementary non-oracle conversations. Agents reach an agreement in 59% and 74% of runs, in which both agents accept specific names, signatures, data shapes, or an ownership split. Agreement is more frequent under *Sync* than *Message*: 72% vs. 46% on contradictory and 90% vs. 57% on complementary. Runs with an agreement pass at 36% vs. 24% and 75% vs. 57%, suggesting that agents need to *agree on shared interface or edit ownership*, rather than merely identify *where* their tasks overlap. In contrast to Khatua et al. (2026) that finds channels "jammed with vague and inaccurate messages," the message content is mostly on target.

Agreements do not prevent conflicting edits. On contradictory tasks, 61% of runs with a concrete agreement still produce a merge conflict, and those runs pass at only 19%. Transcripts show agents agreeing on a design but then both editing the same function, or one agent simply mirroring the other's implementation. Their patches can still *conflict with overlapping changes* even when the agents aim to implement the *same design*.

Knowing the partner's task clarifies edit ownership. The communication oracle reveals the partner's requirement. On complementary tasks, 91% of oracle conversations reach agreement and only 6% produce merge conflicts. All 48 contradictory oracle conversations reach a concrete agreement, 85% assign ownership of the contested code, and merge conflicts fall from 65% to 15% of runs. In the contradictory oracle runs, assigning an owner to the contested code is associated with a higher pass rate of 61%, compared with 29% without one. In 15% of runs, one agent implements both requirements while the other withdraws. However, we note it is also the *least scalable* strategy in reality because with n agents it is infeasible for one agent doing all the work at company scale.

K Additional Related Work

Communication between Agents. DeLM uses verified shared context to help coding agents reuse discoveries across attempts to resolve the same software issue (Mao & Mirhoseini, 2026). AgentRadio supports asynchronous communication among agents investigating different parts of a code-comprehension question, with source modification prohibited in its evaluation (Ren et al., 2026). Our study examines whether communication helps agents coordinate *conflicting* implementation decisions across *independently specified requirements*. This problem setting requires agents to agree on shared interfaces or reconcile conflicting behavioral requirements, and we analyze the tradeoff between integration correctness and execution cost.

When do multiple agents help? Kim et al. (2026) evaluate 260 configurations of different language models and coordination architectures across six benchmarks and find that collaboration benefits depend on task structure and can diminish as individual agents become more capable. A complementary analysis by Cemri et al. (2026) identifies 14 failure modes in multi-agent LLM systems involving system design, inter-agent misalignment, and task verification. These studies characterize broad performance trends and recurring failures, while our study examines a particular coordination setting when separate coding assignments are distributed across multiple agents on the same codebase and result in conflicts.

L DETAILS

L.1 REDIS DOCUMENTATION

https://redis.io/docs/latest/

L.2 CODEX CONFIGURATION

https://github.com/openai/codex, version 0.150.1, medium reasoning effort.

L.3 QWEN3.7 FLASH

Qwen3.7 Flash via OpenRouter: https://openrouter.ai/qwen/qwen3.7-flash.

L.4 MERGE RESOLVER

https://huggingface.co/CooperBench/Qwen2.5-Coder-0.5B-Merge-Resolver.

L.5 OPENROUTER PRICING

OpenRouter Pricing: https://web.archive.org/web/20260924162009/https://openrouter.ai/anthropic/claude-opus-4.8

Cross-model token accounting. The effective-token formula counts fresh input, cache-read, and output tokens; cache-write (cache-creation) tokens are not part of it. This matters only for the cross-model comparison: Claude Code working with OpenRouter does not report cache-write tokens for the open-weight models, so to keep the input term identical in meaning across models we count input tokens only (excluding cache-creation) for every model, including Claude Opus 4.8. The input term for single-model Claude Opus 4.8 results include cache-write tokens.

L.6 CLAUDE CODE VERSION

Claude Code version 2.1.269 : https://code.claude.com/docs/en/changelog#2-1-269

L.7 OPENROUTER CLAUDE CODE INTEGRATION

Following OpenRouter's Claude Code integration recipe: https://web.archive.org/web/20260924134735/https://openrouter.ai/docs/cookbook/coding-agents/claude-code-integration

L.8 DEEPSEEK V4 FLASH

https://openrouter.ai/deepseek/deepseek-v4-flash-0731

L.9 GLM 5.3 FLASH

https://openrouter.ai/z-ai/glm-5.3-flash

M PROMPTS

Every scenario in Section 3.2 is assembled from a *base task prompt* with an optional scenario-specific *overlay* appended. Placeholders in double braces are filled per instance and scenario: {{PROBLEM}}, {{PROBLEM_A}}, {{PROBLEM_B}} are the requirement statements; {{AGENT}}/{{PEER}} and {{PEER_PROBLEM}} identify the partner and its requirement in the communication oracle; {{SUBAGENT_MODEL}} is the model the orchestrator must spawn; and

{{TEST_FILES}} lists the gold-test files in the gold-test scenarios. The mapping from scenario to prompt is:

Scenario	Prompt (base + overlay)
Oracle	base (both requirements), Listing 2
Orchestrator	orchestrator prompt, Listing 3
Sequential / Reentrant	base (single) + hand-off overlay, Listing 1, 6
Parallel unaware	base (single); no overlay
Parallel aware / message / sync	base (single) + coordination overlay, Listing 4
Communication oracle	communication-oracle prompt, Listing 5
Oracle + gold tests	base (both) + gold-test overlay, Listing 7
Parallel + gold tests	base (single) + gold-test overlay, Listing 7

Base task prompts. The single-requirement base (used by every parallel and hand-off scenario, one requirement per agent) and the two-requirement base (used by the single-agent oracle):

Listing 1: Base task prompt, single-requirement form (one agent, one requirement).

```
You are working in a git repository. Resolve the following requirement in the codebase. Make
     the changes directly to the files.

## Requirement

{{PROBLEM}}
```

Listing 2: Base task prompt, oracle form (one agent, both requirements).

```
You are working in a git repository. Resolve both of the following requirements in the
     codebase. Make the changes directly to the files.

## Requirement A

{{PROBLEM_A}}

## Requirement B

{{PROBLEM_B}}
```

Orchestrator. Replaces the base prompt for the orchestrator scenario; the acting agent delegates both requirements to two subagents and does no implementation itself.

Listing 3: Orchestrator prompt.

```
You are the orchestrator of a small team working in a git repository. There are two
     independent requirements to resolve, A and B. Your job is to coordinate the work, not to
     do it.

Hard rules:

- You MUST delegate all implementation to exactly two subagents -- one per requirement. Spawn
     two subagents and have each one implement its assigned requirement in full.
- You MUST NOT implement anything yourself. Do not edit, create, or fix any repository file
     directly. Every change to the codebase must be made by a subagent, not by you.
- Each of the two subagents MUST run on the model `{{SUBAGENT_MODEL}}`. Spawn them with that
     model.
- Your role is limited to: reading enough context to plan, deciding the assignment,
     dispatching the two subagents, and confirming their work is integrated into the working
     tree.

You are free to decide how to run this: which subagent takes which requirement, what
     instructions and context you give each, and whether you dispatch them concurrently or one
      after the other. The only fixed constraints are the three rules above -- two subagents,
     one per requirement, and no hands-on work by you.

## Requirement A

{{PROBLEM_A}}

## Requirement B

{{PROBLEM_B}}

When both subagents have finished, make sure the repository's working tree contains the
     combined result of both requirements.
```

Parallel coordination overlays. Appended to the single-requirement base. *Unaware* adds no overlay. *Aware* states a peer exists but gives no channel. *Message* and *Sync* expose the same send/receive interface, framed as free-form messaging or as explicit synchronization points, respectively.

Listing 4: Parallel coordination overlays (aware, message, sync). Unaware appends nothing.

```
=== aware ===
Another agent is working in this same repository at the same time on a related requirement.
     You will not see their changes while you work.

=== message ===
Another agent is working in this same repository at the same time on a related requirement.
     You can exchange messages with them: run `./send-msg "your message"` to send, and `./recv
     -msg` to read any new messages from the other agent. Use this to share intent, interfaces
     , and assumptions. Check `./recv-msg` periodically, at least before you make interface
     decisions and before you finish.

=== sync ===
Another agent is working in this same repository at the same time on a related requirement.
     You can exchange messages with them: run `./send-msg "your message"` to send, and `./recv
     -msg` to read any new messages from the other agent. Treat the work as a sequence of sync
      points: before starting implementation, send your plan and the interfaces you intend to
     define, then run `./recv-msg` to read theirs; after each significant change, send it and
     run `./recv-msg` to reconcile; before finishing, confirm your assumptions against theirs.
      Do not finalize an interface decision without running `./recv-msg` first.
```

Communication oracle. Extends parallel messaging by giving each agent its partner's requirement verbatim together with an instruction to jointly solve both.

Listing 5: Communication-oracle prompt.

```
You are agent {{AGENT}}. Another agent, {{PEER}}, is working in this same repository at the
     same time on the following requirement:

{{PEER_PROBLEM}}

You will not see their changes while you work. Both sets of changes are merged afterwards, and
      both requirements must hold after the merge. You can exchange messages with them: `./
     send-msg "text"` sends, `./recv-msg` prints new messages from the other agent, and `./
     recv-msg --wait` waits up to 90 seconds for one. Please communicate to jointly solve your
      respective requirements as efficiently as possible.
```

Sequential and reentrant hand-off overlays. Appended to the single-requirement base for the second and later turns, varying whether the acting agent sees only the current file state (*repo*) or also the earlier agent's edit history (*hist*).

Listing 6: Hand-off overlays (repo-only and with-history).

```
=== repo (current file state only) ===
Another agent has already worked in this repository before you. You are picking up the current
      state of the files as they left it. Continue from that state.

=== hist (file state plus edit history) ===
Another agent has already worked in this repository before you. You are picking up the current
      state of the files as they left it, and their edit history is available in the git log (
     use `git log` and `git diff` to see what they changed and why). Continue from that state.
```

Gold-test access overlay. Appended in the *oracle + gold tests* and *parallel + gold tests* scenarios, letting an agent run the gold tests for its own requirement(s) while it works.

Listing 7: Gold-test access overlay.

```
The tests that define "done" for your requirement(s) are provided in the `_gold_tests/`
     directory, and you can run them at any time with `./run-tests`. This applies your current
      changes in a clean copy of the project, runs those tests, and prints which pass and fail
     . Use it to check your work, and make sure they pass before you finish.

Your tests:
{{TEST_FILES}}

(`_gold_tests/` and `./run-tests` are harness scaffolding, excluded from your submission -- do
      not edit them or include them in your changes.)
```