CooperBench: Why Coding Agents Cannot be Your Teammates Yet

Arpandeep Khatua[1*], **Hao Zhu**[1*],

Peter Tran[2**], **Arya Prabhudesai**[2**], **Frederic Sadrieh**[2**], **Johann K. Lieberwirth**[2**],

Xinkai Yu[1], **Yicheng Fu**[1], **Michael J. Ryan**[1], **Jiaxin Pei**[1], **Diyi Yang**[1]

[1]**Stanford University** [2]**SAP Labs US** [*]**Equal Contribution** [**]**Equal Contribution**

`https://cooperbench.com`

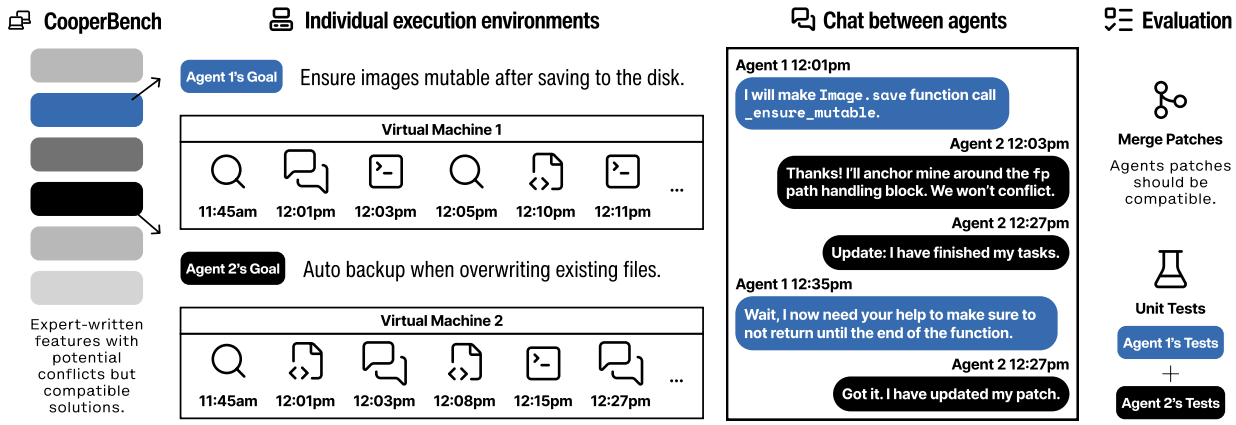


Figure 1 | The CooperBench benchmark draws tasks for two agents from a pool of features with potential conflicts. The agents execute the tasks in their individual environments, communicating in real time to coordinate. Success is measured by whether the resulting code changes by both agents are compatible and pass the requirements for both features.

Resolving team conflicts requires not only task-specific competence, but also social intelligence to find common ground and build consensus. Similarly, as AI agents increasingly collaborate on complex work, they must develop coordination capabilities to function as effective teammates. Yet we hypothesize that current agents lack these capabilities. To test this hypothesis, we introduce **CooperBench**, a benchmark of over 600 collaborative coding tasks across 12 libraries in 4 programming languages. Each task assigns two agents different features that can be implemented independently but may conflict without proper coordination. Tasks are grounded in real open-source repositories with expert-written tests. Evaluating state-of-the-art coding agents, we observe **the curse of coordination**: agents achieve on average **30% lower success rates** when working together compared to performing both tasks individually, across the full spectrum of task difficulties. This contrasts sharply with human teams, where adding teammates typically improves rather than diminishes productivity. Our analysis reveals three key issues: (1) communication channels become jammed with vague, ill-timed, and inaccurate messages; (2) even with effective communication, agents deviate from their commitments; and (3) agents often hold incorrect expectations about others' plans, observations, and communication. Besides these issues, through large-scale simulation, we also observe rare but interesting emergent coordination behavior between agents including role division, resource division, and negotiation. Our research not only presents a novel benchmark for collaborative coding, but also calls for a research shift from pursuing individual agent capability to developing *social intelligence*: the ability to understand others, communicate effectively, and coordinate actions.

1. Introduction

Most achievements in modern civilization arise from individuals working cooperatively, from the construction of cathedrals to the development of open-source software (Raymond, 1999; Woolley et al., 2010). In human societies, such cooperation relies on social intelligence: the ability to communicate intentions, understand others' goals, and negotiate mutually compatible solutions (Humphrey, 1976). This capability is often viewed as what makes us uniquely human and the basis of human thinking (Tomasello, 2014). As we deploy AI agents in cooperative settings, whether strong individual capabilities translate to effective cooperation with either humans or agents remains an open question. In this paper, we empirically demonstrate that for current AI systems, there is a curse of coordination: *agent cooperation perform much worse than a single agent given the same total workload.* This deficit presents a fundamental barrier to deploying AI systems that can work alongside humans or other agents. We theorize that at a fundamental level, effective human–AI and agent–agent cooperation rely on the same coordination abilities.

Glossary

Cooperation: When two or more agents/humans work together towards a shared goal, where an agent may altruistically help another achieve things outside their original responsibility.

Collaboration: When two or more agents/humans work together towards a shared goal.

Coordination: The capability to act and communicate in accordance with other agents/humans.

Existing research on automating human tasks and multi-agent systems largely sidesteps this challenge by either providing more scaffolds (Fourney et al., 2024a; Pan et al., 2025; Zhang et al., 2025b; Zhuge et al., 2024), enforcing strict workflows (Cheng et al., 2025; Hong et al., 2023; Nguyen et al., 2024), or providing active supervision and verification (Huang et al., 2025; Xiang et al., 2025; Zheng et al., 2025). These systems rely on developer- or user-provided scaffolding to manage coordination, which limits flexible cooperation and places additional burden on humans.

We present CooperBench, the first benchmark designed to measure how well agents can cooperate when handling individual tasks with potential conflicts. Considering software engineering as a realistic domain where humans typically need to navigate work in a team (Purna Sudhakar et al., 2011), our benchmark offers verifiable evaluation for the success of agent cooperation. As illustrated in Fig. 1, CooperBench comprises 652 tasks constructed from 12 popular open-source libraries across Python, TypeScript, Go, and Rust. Eight co-authors of this paper with real-world software engineering backgrounds created new features, unit tests, and ground-truth code for these libraries, ensuring high-quality and realistic task design.

In CooperBench, each task assigns each agent a feature to be implemented based on the same repository state. Conflicts are intentionally embedded at the code level, as the assigned features are logically compatible but require agents to modify overlapping or interdependent code. For example, in Fig. 1, one agent implements image mutability in the serialization process while another adds backup functionality to the same process. Without understanding each other's goals, plans, and expectations, their solutions may introduce incompatible changes. This mirrors real-world software development where coordination failures stem from insufficient mutual understanding. CooperBench enables us to investigate three research questions:

RQ1: How well can agents cooperate with each other? (§4)

RQ2: What role does communication play in agent-agent cooperation? (§5)

RQ3: What coordination failures do agents exhibit? (§6)

Through evaluating state-of-the-art coding agents on CooperBench, we observe *the curse of coordination*: `GPT-5` and `Claude Sonnet 4.5` based agents achieve only 25% with two-agent cooperation on CooperBench, which is around 50% lower than a "Solo" baseline which uses one agent to implement both features.

Diving deeper into the coordination failures, we identify three key issues. First, communication channels become jammed with vague, ill-timed, and inaccurate messages where agents fail to respond to direct questions, send messages that arrive too late to inform decisions, or flood channels with repetitive status updates that lack actionable detail. Second, even with effective communication, agents deviate from their commitments. They make unverifiable claims about code state, ignore agreed-upon integration points, and break explicit promises. Third, agents hold incorrect expectations about their partner's plans, observations and duplicate work despite warnings and overwrite changes they believe will merge cleanly (§6).

Besides failures, we are excited to report emergent coordination behaviors which often lead to the success of the CooperBench tasks. These coordination behaviors are rarely performed by the agents, but through our large-scale simulation, we uncover three major categories of them: role division, resource division, and negotiations (§6.4). These examples hint at a path of coordination capability acquisition through reinforcing success on CooperBench.

We contribute both a novel understanding of what agents need to become effective teammates and a practical benchmark for measuring progress. Our open-sourced CooperBench platform enables researchers and practitioners to evaluate and improve cooperative coding agents.

2. CooperBench Benchmark

CooperBench seeks to satisfy the following desiderata: (1) *Realism*: the tasks should be reasonable for a software development team to work on at a given repository state. (2) *Conflict potential*: the agents' scopes should overlap with each other so that they need to coordinate well to avoid potential conflicts. (3) *Verifiable*: the success of the tasks can be evaluated with a pipeline that is deterministic and interpretable. These three desiderata provide a basis for accurately measuring the real-world cooperation capabilities of agents.

2.1. Task space

Task Each task consists of a repository state, two features, and two corresponding sets of unit tests. The two features are drawn from a pool of features (like the one illustrated in Fig. 2) that can be simultaneously implemented on the given repository state. The patches from the two agents are merged and evaluated. Each agent's goal is to get their assigned feature im-

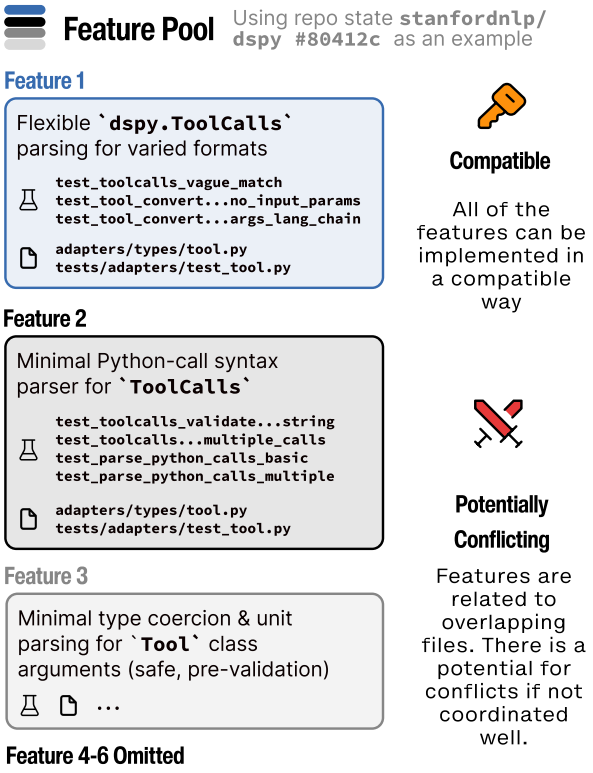


Figure 2 | An example feature pool based on DSPy GitHub repository. This feature pool has 6 features which can be implemented compatibly based on the repository state, but without coordination agents could conflict with each other.

plemented in the merged patch.[1] Based on a pool of n features, there will be $\binom{n}{2}$ tasks when we are evaluating agents self-play, and double the number when we evaluate two different agents cooperate with each other. Note that agents can only view their own features. For example, in Fig. 2, there are 6 features in this pool, which produces 15 tasks for evaluating `GPT-5` agents cooperating with each other. If we want to evaluate how well `GPT-5` agents cooperate with `Claude Sonnet 4.5` agents, we will have 30 tasks drawn from this pool. In CooperBench we have 34 such features pools.

Features In this paper, we use `features` to denote desirable changes to the codebase that implement missing functionality, fix existing bugs, or both. As illustrated in Fig. 2, each `feature` is described in a markdown file, which includes a title, description, examples, and a list of files which may be relevant. For each feature, we write a list of unit tests without the help of coding assistants to ensure accurate evaluation of the implementation. In addition, we write a ground-truth solution to understand the potential conflicts between features and to verify that the given feature can be implemented on the repository and pass the unit tests. The tests and the ground-truth solution is not provided to the agents to prevent test leakage.

Task composition For each repository state, we create a pool of feature candidates. These features are *compatible* and *potentially conflicting*. "Compatible" means the features can be implemented jointly. To verify this, we produce a joint ground-truth solution of all features in the pool, which passes all individual unit tests. "Potentially conflicting" means the features have overlapping code logic changes that influence each other. In our dataset, 77.3% of tasks have conflicting ground-truth solutions. As a result, CooperBench tasks are not adversarial, but still require the capability to cooperate under conflicts by communicating individual goals, understanding others' plans, and negotiating mutually compatible solutions.

Action space Agents can take two kinds of actions in real time: the *communication tool* and *computer-use tools*. The communication tool allows agents to send open-ended natural language messages to each other, and the computer-use tools include file and terminal operations. In our paper, we limit the computer-use tools to local operations to control the experiments. In the future, researchers could consider GUI and browser-based actions to expand the tasks the agents can take. Both agents can use these tools at any time, without synchronizing their turns with each other. This not only raises the flexibility of agents, but also poses challenges for agents to timely communicate and execute commands. In our benchmarking process, we use cloud virtual machines for agents to ensure isolated workspaces and sufficient resources. We set an upper-bound number (100)[2] of actions an agent can take to complete the tasks.

2.2. Evaluation pipeline

Cooperation is hard to evaluate, but we make the product of the cooperation verifiable. CooperBench evaluates tasks based on two criteria: (1) *compatible solutions* and (2) *implementation correctness*.

Solution compatibility After the two agents complete execution, we attempt to merge their resulting patches using `git merge-file -L patch_1 -L repo_state -L patch_2`. This operation captures whether the independently produced solutions are structurally compatible. In practice, some merge failures arise from superficial differences such as formatting or indentation styles (e.g., K&R versus Allman) rather than substantive conflicts. To avoid treating such cases as coordination failures, we train a small coding model (`Qwen 3 Coder 1.5B`; Yang et al. 2025) on synthetic examples to resolve

[1] Agents have the freedom to redivide the two features as long as the merged patch implements both features. Agents perform this kind of coordination occasionally well. Check out §6.4 for concrete examples.

[2] We do not observe performance gains on our tasks from raising this number.

trivial merge conflicts when standard merging fails. This step ensures that the compatibility check reflects semantic agreement between solutions rather than low-level stylistic discrepancies, while leaving the overall cooperation score largely unaffected (App. § B). If even the coding model cannot produce a patch without conflicts, the agents both fail the tasks.

Implementation correctness If we successfully merge the two patches into the repository state, we run both sets of unit tests on the merged codebase. As mentioned before, we do not restrict agents to only finish their own work. If they can coordinate well, they can divide their two features in a different way as long as the merged solution can pass the two features' tests. This evaluation pipeline ensures a rigorous evaluation of the cooperation outcome.

2.3. Dataset Construction

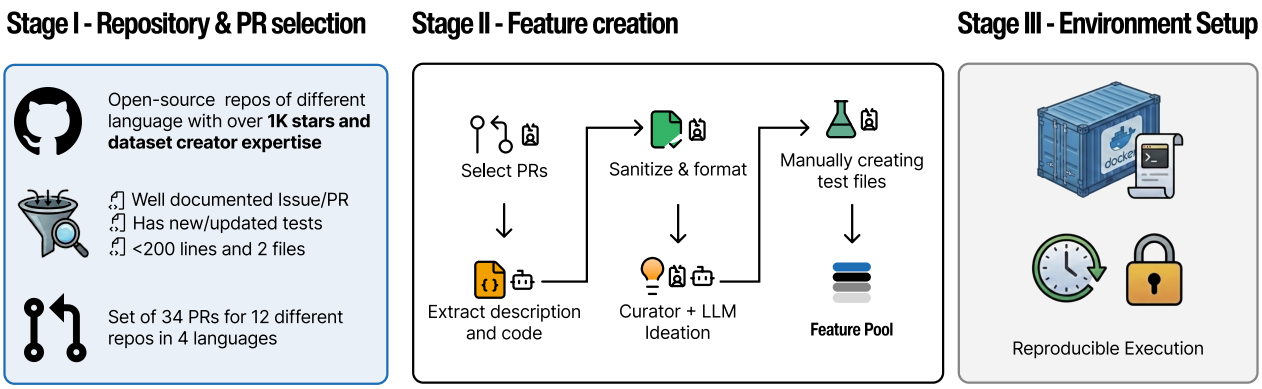


Figure 3 | The CooperBench construction pipeline. Each task is carefully engineered by domain experts to ensure conflicts are realistic, resolvable, and representative of production software development challenges.

CooperBench is constructed through a three-stage pipeline that grounds tasks in real software development and enables controlled evaluation of coordination (Fig. 3). To create the pools of features, we start from real-world feature implementations and proceed as follows: (Stage I) we write *anchor features* drawn from popular repositories, each of them is a slight modification of a real pull request (PR) authored by human contributors; (Stage II) for each anchor feature, we expand the pool by introducing a family of *adjacent features* authored by human annotators, representing plausible alternative features that could realistically co-occur; and (Stage III) we validate the compatibility of each feature pool by executing and testing all feature combinations in a controlled environment to rule out intrinsically incompatible specifications.

Stage I: Repository and PR Selection In the first stage we select twelve actively maintained open-source repositories spanning Python, TypeScript, Rust, and Go. Each repository exceeds one thousand GitHub stars and does not appear in SWE-Bench (Jimenez et al., 2023) or Multi-SWE-Bench (Zan et al., 2025), reducing data contamination risk. Selection is guided by curator expertise so that each repository is assigned to an author familiar with its architecture and development practices. We extract PRs that meet strict inclusion constraints: clear feature description, code+tests, feature addition, bounded change size, and robust tests. Appendix A provides full selection details and thresholds, and App. Tab. 3 summarizes the repository distribution.

Stage II: Feature Extraction and Augmentation In the second stage, we convert each selected PR into a feature pool containing one anchor feature and multiple synthetic adjacent features. We sanitize and rewrite original PR descriptions into self-contained specifications to prevent information

leakage. Curators author adjacent features to plausibly co-occur and to create natural overlap without adversarial specifications (with LLM-assisted ideation). Appendix A provides full details on adjacent-feature design, manual test writing, and gold-solution validation. All features derived from the same base commit constitute a feature pool with two to 12 features. To ensure the compatibility among all features in a pool, we construct a single gold patch that jointly implements all features in each set and passes all associated tests.

Stage III: Environment and Reproducibility The final stage provides a deterministic execution environment for evaluating agents. Each task set includes an automated setup script that clones the repository at the exact base commit, installs dependencies, and executes the full test suite to verify the environment. To ensure consistent behavior across hardware and operating systems, we additionally provide containerized environments that encapsulate the complete repository state and all runtime dependencies. These environments guarantee reproducible execution and isolate agent behavior from external variability, enabling reliable measurement of coordination performance through the evaluation pipeline described in §2.2.

Dataset composition and feature-complexity statistics are reported in App. A. Together, these findings demonstrate that CooperBench features are individually tractable and realistic, ensuring that the benchmark's primary challenge arises from coordinating partially overlapping implementations rather than from executing unusually complex or oversized programming tasks.

3. Experiment Settings

CooperBench allows us to study the following research questions. First, how well can current state-of-the-art foundation models cooperate with each other when they are used in coding agents? Second, do agents use the communication channel effectively for coordination? And, what are the reasons why agents fail or succeed on CooperBench?

In order to evaluate models fairly, we create an agent framework incorporating leading open-source coding agent framework `OpenHands (v0.54)` (Wang et al., 2024b). The two agents perform their own work in their respective docker-based containers without interruption from another agent. Since OpenHands was not designed as a framework which performs multi-agent cooperation, we created a communication tool (§2.1) using an SQL database for message passing. This communication tool supports message sending action. When an agent sends a message to another agent, the other agent will immediately receive it, and include it in the prompt of the next step. This communication setting achieves both real-time communication and asynchronous execution. We open-source this framework to not only ensure reproducibility of our experiments, but also provide a starting point for researchers to build multi-agent cooperation systems which can perform multiple tasks and resolve conflicts.

However, note that CooperBench does not tie with the agent framework or the communication tool. In this paper, we are more concerned with foundation models' intrinsic capability to cooperate, so we do not compare different agent frameworks or creative methods to enhance coordination. In the future, researchers should use CooperBench to compare different models, different frameworks, and different combinations as well. We especially encourage researchers to develop novel frameworks or to train agents to achieve higher Coop scores or to close the Solo-Coop gaps (§4) on CooperBench. Similarly, we encourage researchers to develop other communication tools, e.g. screen sharing, to expand the communication bandwidth or reduce the communication noises.

We evaluate the performance of five language models, both closed-source ones, and open-source

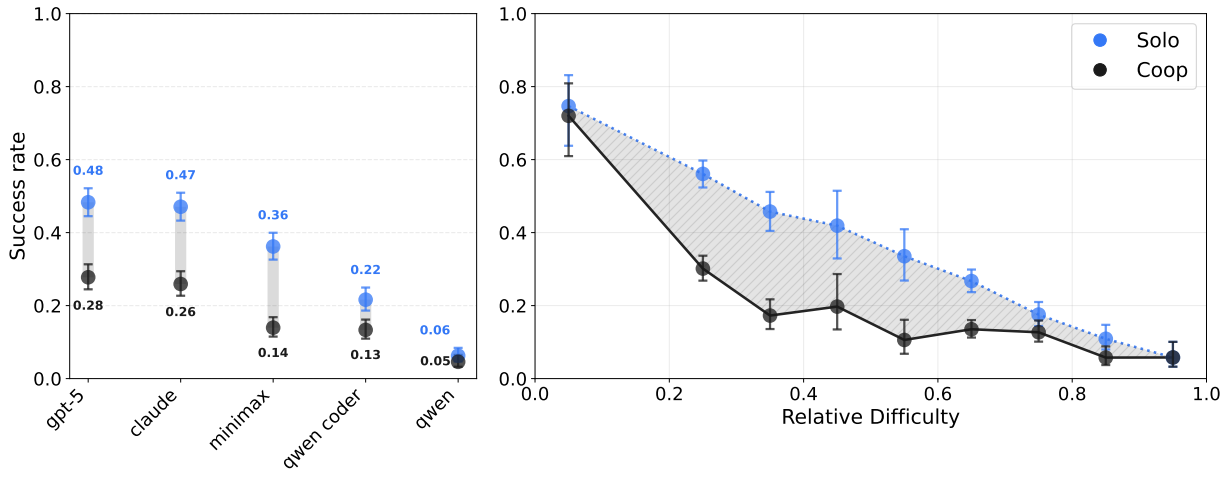


Figure 4 | **Left:** Under Coop setting, agents with different foundation models perform significantly worse than how they perform under Solo setting, except for `Qwen3-30B-A3B-Instruct-2507`, which performs bad under both settings. This Solo-Coop gap is what we call the "coordination gap". **Right:** The relationship between tasks' *technical* difficulties and Solo-Coop gap. The shaded area has a large middle section which shows that the coordination gap is larger for middle-level tasks than for tasks which are extremely easy or difficult.

ones: `GPT-5`, `Claude 4.5 Sonnet`, `MiniMax-M2`, `Qwen3-Coder-30B-A3B-Instruct`, and `Qwen3-30B-A3B-Instruct-2507`. We serve the two Qwen models via vLLM[3], GPT-5 and Minimax models via their respective official API, and the Claude model through GCP.

4. How well are agents able to cooperate with each other?

In CooperBench, each of the two agents are assigned a feature to implement, which will be called the Coop setting to distinguish from the Solo baseline. In the Solo baseline, the two tasks are assigned to one agent. For humans, teams should perform better or faster than individuals, which is the bottom line for cooperation to be considered as functional. We hypothesize for agents, the advantage of cooperation is overwhelmed by their incapability to coordination. This should lead to a "coordination gap": two agents perform worse than one agent for the same workload.

The curse of coordination. As shown in Fig. 4 (Left), across all models, success rates under the Coop setting is consistently lower than those under Solo settings, which means when two agents need to coordinate between them, they perform even worse than one agent "solo"ing the two features. This coordination gap is as large as 50% in the leading models: `GPT-5`, `Claude Sonnet 4.5`, and `Minimax M2`. Qwen models have smaller gaps, but their Solo setting score is much lower as well. All error bars in Fig. 4 are 95% *Wilson* confidence intervals computed over task sets (App. C).

Mid-difficulty crisis. As shown in Fig. 4 (Right), the gap between the two settings is larger and more significant on the tasks with middle-level technical difficulty than on the ones which are too easy or too hard. Here we stratify tasks by *relative difficulty*. For each task pair t, we define a raw difficulty score $d(t) = 1 - \frac{1}{|\mathcal{M}|}\sum_{m\in\mathcal{M}} \text{Solo}_m(t)$, where $\text{Solo}_m(t)$ denotes model m's Solo success on t.

[3] `https://vllm.ai/`

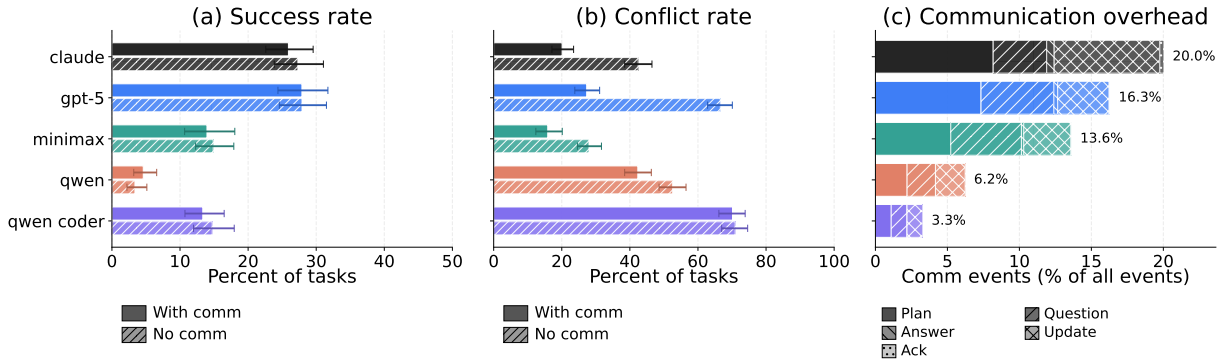


Figure 5 | **(a)** Effect of inter-agent communication on cooperation success or lack thereof. All agents fail to use communication for improving cooperation success. **(b)** Communication substantially reduces naive merge conflicts across all models. **(c)** Communication overhead as a percentage of all execution events, broken down by message type. Models that communicate more (e.g., `Claude Sonnet 4.5`, `GPT-5`) show larger reductions in conflict rate.

For visualization, we linearly rescale $d(t)$ to $\tilde{d}(t) \in [0, 1]$ using the minimum and maximum $d(t)$ values in the benchmark. We bucket tasks by $\tilde{d}(t)$ and report success rates as a function of $\tilde{d}(t)$ for both Solo and Coop. This result points out that agents struggle to balance the two pressures for technical difficulty and cooperation difficulty. When tasks are too easy, agents could spare more effort for coordination, but when tasks are getting harder, agents cannot effectively coordinate.

Scaling the number of cooperating agents. Our hypothesis is that increasing the number of agents in the same cooperative workspace exacerbates coordination overhead (e.g., more context to track and more opportunities for inconsistent plans), leading to lower end-to-end success. To probe this directly, we run a small scale experiment using 46 tasks from 3 separate task sets where we scale the number of concurrently cooperating agents from 2 to 4 while keeping the cooperative setting fixed. We observe a monotonic decline in success as the number of agents increases. Specifically, performance drops from 68.6% with 2 agents to 46.5% with 3 agents and further to 30.0% with 4 agents on the tasks, reinforcing the "curse of coordination" beyond the 2-agent setting.

5. What is the role of communication in agent-agent cooperation?

In CooperBench, the communication tool we provide is the only channel agents could use to coordinate with each other. Can agents effectively use it? We hypothesize that although agents might actively use the tool, their communication might be far from effective or efficient. To evaluate this, we compare with a baseline setting, where the communication tool is banned, i.e. "no comm".

Communication does not lead to better cooperation. As shown in Fig. 5 (a), none of the models effectively leverage communication tool to achieve higher cooperation success. The difference between "with comm" and "no comm" settings is not statistically significant. This shows that existence of the communication tool does not help coordination. Does this mean agents are not using them? We quickly negate this question through examine the usage and the conflict rate.

Communication reduces merge conflicts. As shown in Fig. 5 (b), communication does significantly reduce the merge conflicts between patches in `Claude Sonnet 4.5`, `GPT-5`, `MiniMax M2`, and `Qwen Instruct`. This shows that agents could leverage the communication tool to reduce overlap in their work, despite that just avoiding conflicts does not warrant cooperation success. Communication also consumes a meaningful share of the agent's action budget. Fig. 5 (c) reports the frequency of all communication speech act types. This result shows that agents spent as much as 20% of the steps in communication, within which planning, questioning, and updating each almost takes up 1/3 of steps. But why this much effort in communication does not translate into better cooperation?

What distinguishes effective communication? To understand why communication helps conflicts but not success, we analyze what *successful* communication looks like. Three patterns emerge.

First, successful agents plan more and question less. Trajectories that avoid conflicts have a Plan:Question ratio of 2.04, compared to 1.31 for conflict trajectories. This suggests that questions are a *symptom* of coordination problems, not a cure. Agents that are already struggling tend to ask more questions, but questioning does not prevent conflicts.

Second, first-turn planning is the strongest predictor. Having a `Plan` message in the very first turn nearly halves the conflict rate (29.4% vs 51.5%). This effect is robust across difficulty levels: in 7 out of 8 difficulty buckets, first-turn planning significantly reduces conflicts, with the effect actually *stronger* for harder tasks (39% reduction at the highest difficulty).

Third, specificity matters. Successful trajectories contain significantly more concrete references: 32.6 line number mentions versus 22.5, and 13.1 file path mentions versus 10.0. Agents that communicate *where* they are editing with specific line ranges successfully avoid overlapping changes.

Spatial vs. semantic coordination. These findings explain why communication helps conflicts but not success. Merge conflicts are fundamentally a *spatial* coordination problem: agents must agree on who edits which lines. The patterns above (early planning, specific line numbers, file paths) all address spatial coordination, and they work.

However, task success requires *semantic* coordination: understanding *what* to implement, not just *where*. Our case study in Appendix I illustrates this gap. Two agents successfully coordinated on line numbers and edit ranges (spatial), yet failed because they never discussed the actual parameter values their implementations should use (semantic). They solved the "formatting" problem of avoiding overlapping edits but not the "design" problem of ensuring compatible implementations.

Repetition, Unresponsiveness, and Hallucination. Beyond the spatial-semantic gap, the communication itself is often flawed. We identify three major communication problems, and show their frequencies in Fig. 6. We automatically detect these patterns using an LLM-as-judge approach with a precision-focused taxonomy; see Appendix F for the full rubric and evidence requirements. Repetition consumes budget without adding constraints a partner can act on, which is consistent with high communication overhead without commensurate gains in end-to-end success. Unresponsiveness breaks the feedback loop when one agent asks for a decision that gates implementation, and incorrectness creates false shared context, such as asserting an interface decision or a completed change that is not actually satisfied. Hallucination results in noises which makes it hard to partners to coordinate under imperfect information.

In this section, we show that the communication tool is heavily used, but *not* properly leveraged by agents for coordination. This shows that agents lack the critical *pragmatics* understanding of

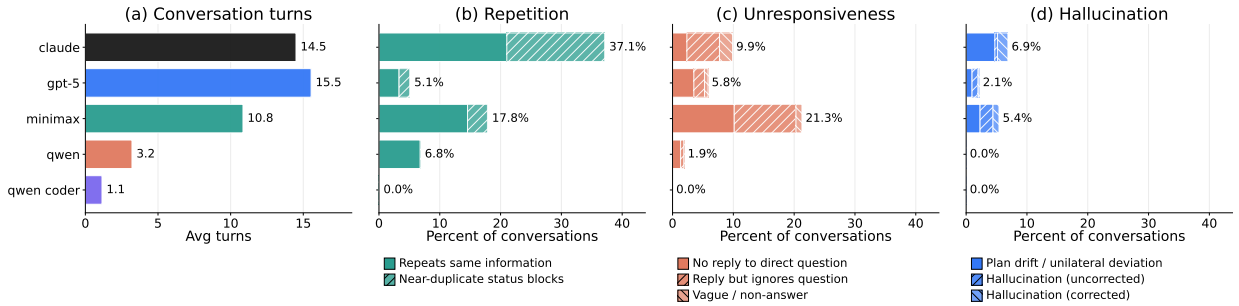


Figure 6 | Break down frequencies of different kinds of communication errors.

language: communication is not just about message passing, but about achieving certain functions through passing the messages. Agents are "talking" a lot, but they cannot achieve their communication goals through communication when communication channel is jammed with repetitions, unresponded questions, or false information.

6. What are the coordination failures that the agents exhibit?

Section 5 showed that communication alone does not improve coordination. Why not? We find that even when agents communicate their plans, they struggle to honor commitments and anticipate partner actions. Coordination failures stem from three capability gaps: *communication* (failing to exchange key information), *commitment* (not following through on promises), and *expectation* (failing to model what partners are doing). We first categorize failures by their observable *symptoms* (§6.1), then identify these underlying *causes* (§6.2).

6.1. Failure Symptoms

We analyze all failed Coop trajectories across all five models on the full dataset. Through iterative qualitative coding, we develop the failure symptom taxonomy shown in Tab. 1. We then use `GPT-5` as an LLM-as-a-Judge to categorize trajectories at scale, yielding the frequency distribution in Tab. 1. The resulting vocabulary provides a structured way to diagnose coordination breakdowns. See App. G for the annotation procedure and human validation.

6.2. Failure Reasons

Symptoms describe *what* went wrong; causes explain *why*. To identify the underlying capability gaps, we manually reviewed 50 failed Coop traces. For each trace, we examined the symptom labels, conversation logs, and merged artifacts to determine why coordination broke down. We grouped root causes into three categories shown in Tab. 2. Unlike symptoms, which can be reliably detected by an LLM annotator, causes require deeper interpretation of the coordination dynamics and are therefore manually assigned.

6.3. Representative examples of capability gaps

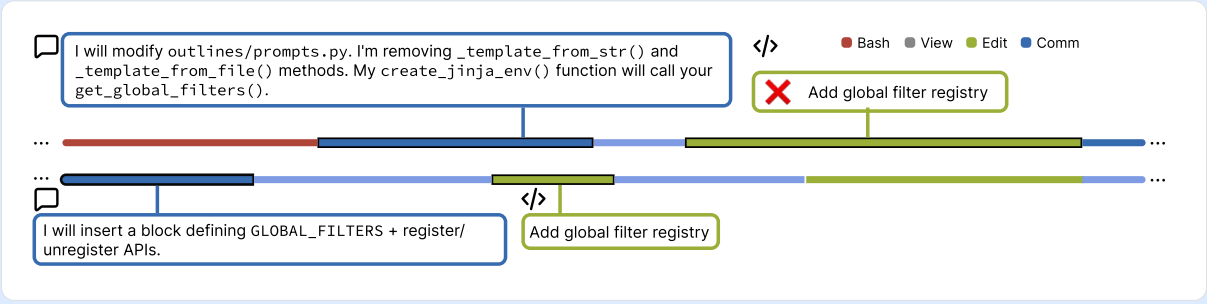
We provide one representative example for each coordination capability gap. Additional symptom-level examples are available in Appendix H.

Table 1 | Coordination failure symptoms. Observable patterns in how coordination breakdowns surface in merged artifacts.

Symptom	Meaning	%
Work overlap	Both agents independently implement the same functionality, duplicating work and overwriting details.	33.2
Divergent architecture	Incompatible design decisions lead to semantic loss even under a clean merge.	29.7
Repetition	Verbose status messages add little new information and reduce signal.	14.7
Unresponsiveness	Direct questions or requests are not answered, breaking the decision loop.	8.7
Unverifiable claims	Agent asserts a change or interface decision without evidence the partner can check (no checkable commitment).	4.3
Broken commitment	Confident completion claims create false shared context when the promised change is absent.	3.7
Dependency access	Missing risk communication leaves agents unable to anticipate merged dependency interactions (e.g., circular imports).	1.7
Placeholder misuse	An explicit integration contract exists but is applied differently than agreed.	1.5
Parameter flow	Ambiguity about a changing interface leaves one agent implementing against an outdated contract.	1.3
Timing dependency	Agents agree on order but fail to communicate an enforceable plan that preserves it after merge.	1.1

Expectation. In the first example, Agent A announces it will modify `prompts.py` and call B's `get_global_filters()`. Agent B states it will insert `GLOBAL_FILTERS` at a specific location. Both agents communicate their plans explicitly, yet the merge fails. The problem is not missing information but failure to *integrate* it. Despite hearing B's plan, A proceeds as if B's code won't exist. This is the most common cause, reflecting a fundamental difficulty in maintaining an accurate model of partner state during independent work.

Expectation: Failure to model partner state

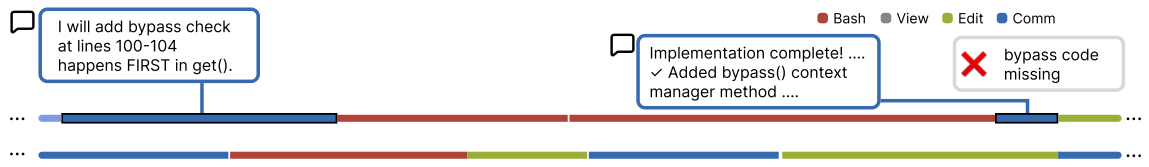


Commitment. In the second example, the agent promises "I will add bypass check at lines 100–104, happens FIRST in get()." Later it claims completion with a checkmark. But after merge, the bypass code is missing. The partner trusted this claim and built on it, but under workspace isolation, trust is all they had. The commitment was *unverifiable*. No pasted signature, no diff, nothing the partner could check without access to the branch.

Table 2 | Coordination capability gaps. Underlying causes inferred through qualitative analysis of failure traces.

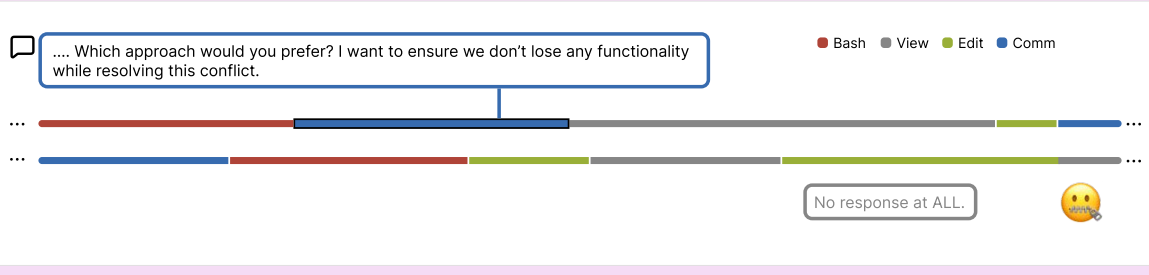
Cause	Definition	%
Expectation	Cases where one agent has clearly communicated what they are doing, but the other agent still treats the situation as if that work is not being done. This reflects a failure to model the state of the other agent's code changes and what that means for the system as a whole.	42
Commitment	Cases where an agent is not doing the things they promised to do. This includes failures to establish or maintain verifiable integration contracts, where agents make commitments but do not follow through on them.	32
Communication	Breakdowns in using language to coordinate. This includes failures in information sharing and decision loops between agents, where agents do not effectively communicate their intentions, questions, or status updates.	26

Commitment: Failure to follow through on promises



Communication. In the third example, Agent A asks a direct question, "Which approach would you prefer?" The response is silence. Without an answer, the coordination loop collapses. A needed a decision to proceed, and without one, both agents continued with potentially incompatible assumptions. Unlike expectation failures (where information exists but isn't integrated) or commitment failures (where promises aren't kept), this is a failure to even establish shared context.

Communication: Breakdown in using language to coordinate



The examples above reveal why coordination, rather than raw coding ability, is often the limiting factor. The common thread is *partial observability*. Each agent acts while holding an uncertain model of its partner's state, edits, and commitments. A merge can be conflict-free yet still embed incompatible assumptions.

These causes manifest through the symptoms in Tab. 1. Expectation failures produce work

overlap and silent overwrites, commitment failures lead to unverifiable claims and broken promises, and communication failures result in unresponsiveness and repetition.

These failures suggest current models lack reliable representations for (i) *partner state* (what the other agent has actually changed), (ii) *checkable commitments* (contracts verifiable after merge), and (iii) *cross-branch integration reasoning* (anticipating how independent patches interact). Coordination requires more than plausible code. It requires *verifiable* and *actionable* constraints for a partner operating under isolation. This explains why prompt optimization yields only marginal improvements (App. D). Most errors stem from coordination challenges, not prompt wording.

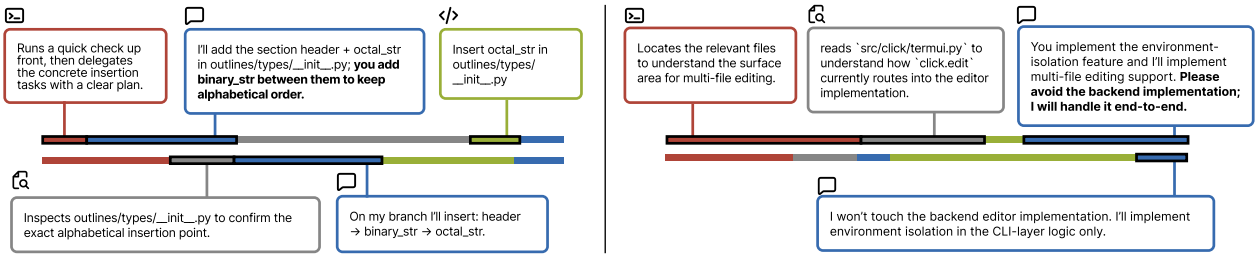
The trust paradox. We hypothesize that a deeper tension underlies expectation failures. Models are trained to be cautious, requiring observable evidence and resisting unverifiable assertions. This is a sensible default for single-agent interactions, where users may attempt to mislead the model. However, collaboration under workspace isolation requires the opposite. Agents must trust partner claims about states they cannot observe. When Agent A reports "I added the handler at line 50," Agent B's instinct is to verify, but verification fails because they are on separate branches. This mismatch between *verification-first* training and *trust-requiring* collaboration may partly explain why agents consistently fail to update their model of partner state despite explicit communication.

Effective collaboration likely requires lightweight mechanisms that turn conversation into verifiable shared state, such as pasted signatures, explicit insertion-point contracts, and integration checks before declaring safety. We now turn to successful cases to see what these mechanisms look like in practice.

6.4. Emergent Coordination Behavior

Among successful runs, we observe coordination patterns that are largely absent from failures. These behaviors are not prompted or scaffolded; they emerge when agents successfully navigate partial observability. What they share is a shift from vague intentions to specific commitments that a partner can verify even without seeing the underlying work. We identify three such patterns.

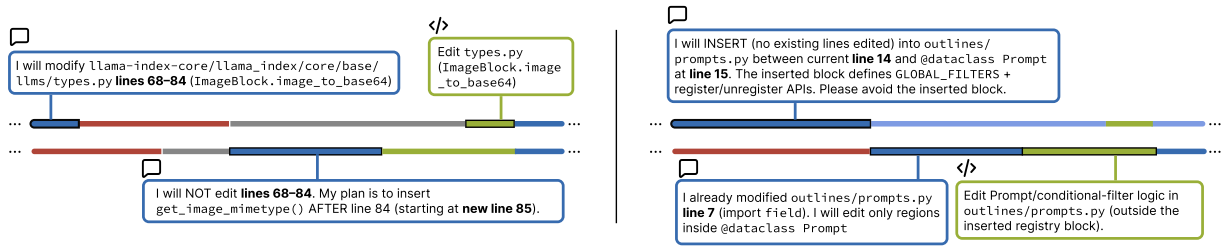
Role division. Agents agree on who handles which part of the task and establish clear boundaries around their scope.



What distinguishes successful role division is mutual confirmation. Under partial observability, a unilateral declaration can easily be missed or misunderstood. When both agents explicitly acknowledge the split, they create verified shared understanding that both sides can rely on during independent work.

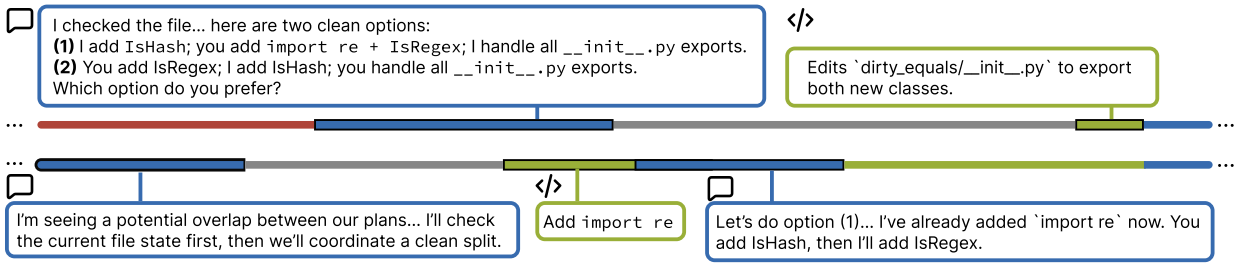
Resource division. Agents avoid collisions by partitioning shared resources, most commonly specific files, code ranges, or *ownership blocks*.

What makes resource division effective is specificity. Vague commitments cannot be verified and thus require trust. Line-level boundaries, by contrast, create safe zones where conflict is impossible



by construction.

Negotiation. Agents resolve conflicting approaches by proposing alternatives and converging on a single plan before acting.



Effective negotiation does cognitive work for both parties. By proposing mutually exclusive options that fully specify what each agent will do, one agent reduces a complex coordination problem to a simple choice. The result is not just agreement on intent but complete action specifications that leave nothing to interpret.

These coordination patterns are rare in our traces but their presence in successful cases suggests that the underlying capability exists. The challenge is not teaching agents new coordination skills but making existing ones reliable.

7. Related Work

Multi-agent LLM systems and tool-using coding agents have advanced rapidly, but reliable collaboration remains unresolved. Prior work largely evaluates task success under engineered interaction structure rather than free-form coordination under partial information.

Multi-agent LLM systems Many frameworks improve performance through structured interaction. CAMEL (Li et al., 2023a) and AutoGen (Wu et al., 2023) use conversation programming; MetaGPT (Hong et al., 2024) and ChatDev (Qian et al., 2024) emulate software organizations; Magentic-One (Fourney et al., 2024b), MAGIS (Tao et al., 2024), and AgileCoder (Nguyen et al., 2024) use explicit orchestrators. Even with such scaffolding, multi-agent systems exhibit high failure rates. Multi-agent configurations degrade performance by 39 to 70 percent relative to single-agent baselines (Su et al., 2025), and failure analyses identify inter-agent misalignment as a major category (Cemri et al., 2025). These findings suggest that externally imposed protocols mask rather than solve the underlying coordination problem. Sotopia (Zhou et al., 2024) provides a general framework for evaluating agents' social intelligence, while our work focus specifically on cooperative coding agents with verified tasks.

Tool-using coding agents such as SWE-agent (Yang et al., 2024), OpenHands (Wang et al., 2025), and Agentless (Xia et al., 2024) achieve strong results on SWE-bench (Jimenez et al., 2024). However,

these evaluations measure single-agent success rather than whether multiple peers can integrate changes without conflict under partial information.

Coordination benchmarks Existing benchmarks span games, embodied tasks, and reasoning. Hanabi (Forkel & Foerster, 2025) and Cicero (, FAIR) test coordination under information asymmetry; MultiAgentBench (Zhu et al., 2025) and Collab-Overcooked (Sun et al., 2025) evaluate LLM collaboration; Tool-RoCo (Zhang et al., 2025a) and RoCoBench (Mandi et al., 2023) assess multi-robot cooperation. In software, SyncBench (Guo et al., 2025) tests divergent understanding and The Collaboration Gap (Davidson et al., 2025) finds that solo-capable models degrade when required to collaborate. These benchmarks typically enforce turn-taking or shared observability rather than testing code integration under workspace isolation. Agent-human collaboration benchmarks such as Co-Gym (Shao et al., 2025), HULA (Takerngsaksiri et al., 2025), and HAI-Eval (Luo et al., 2025) study settings where humans arbitrate. We instead study whether agents can coordinate autonomously.

Theory of Mind evaluation Effective coordination requires modeling partner beliefs and intentions, which commonly referred to *Theory of Mind* (Premack & Woodruff, 1978; Rabinowitz et al., 2018; Zhu et al., 2021). ToMBench (Chen et al., 2024), FANToM (Kim et al., 2023), and SoMi-ToM (Fan et al., 2025) evaluate theory of mind in LLMs, finding substantial gaps versus human performance. ToMSWE (Zhou et al., 2025) tries to build coding agents which can infer users' Theory of Mind. Studies of cooperative games (Li et al., 2023b) and Generative Agents (Park et al., 2023) show emergent social behaviors but also challenges translating these to verifiable collaborative work.

We isolate free-form coordination as the central object of evaluation. CooperBench assigns two agents partially overlapping features on a shared codebase while isolating their workspaces and restricting coordination to natural language. Unlike benchmarks that impose interaction structure or measure outcomes alone, we evaluate through coordination failures such as redundancy, inconsistent assumptions, and semantic breakage. We demonstrate the curse of coordination in a controlled setting with verifiable code integration, pointing to social intelligence as the bottleneck for effective agent teamwork.

8. Conclusion and Future Work

In a future where agents team with humans in high-stakes domains (Kim et al., 2025), accelerate science and technology research (Gottweis et al., 2025), and empower creative endeavors (Waikar, 2021), it is hard to imagine how an agent incapable of coordination would contribute to such a future, however strong their individual capabilities.

Our work demonstrates that coordination, not raw coding ability, is a central bottleneck for multi-agent software development. Through CooperBench, we show that frontier models like `GPT-5` and `Claude Sonnet 4.5` achieve only 25% success when two agents collaborate, roughly half the success rate of a single agent performing the same workload. This *curse of coordination* stems from three capability gaps: agents fail to communicate actionable information, deviate from their own commitments, and hold incorrect expectations about their partners.

Yet coordination is not beyond reach. In successful traces, we observe emergent behaviors such as role division, resource division, and negotiation that turn vague intentions into verifiable commitments. These patterns are rare but their presence suggests the underlying capability exists; the challenge is making it reliable. With multi-agent training methods, e.g. Sotopia-π (Wang et al., 2024a; Yu et al., 2025), we can expect these emergent behaviors to be reinforced through the success of cooperation.

Our findings open several directions: (1) training objectives that reward coordination under partial observability, (2) lightweight protocols for verifiable commitments (e.g., shared signatures, insertion-point contracts), and (3) richer communication channels such as screen sharing to expand the modality beyond text. We release CooperBench as an open benchmark to measure progress on these fronts.

Although we focus on software development, our findings generalize to any domain involving role and resource conflicts under partial observability. We expect that the lack of social intelligence, the ability to understand others, communicate effectively, and coordinate actions, will remain a fundamental barrier limiting the real-world deployment of agents as teammates until these capabilities are explicitly developed.

Acknowledgments

This research is supported in part by grants from ONR grant N000142412532, NSF grant IIS-2247357, DSO National Laboratories (DSO), and support from SAP. We thank Google Cloud Platform and Modal Platform for their credits. We thank Yutong Zhang, Gavin Li, Hannah Cha, John Yang, Yijia Shao, Jenn Wang, Abe Hou, and all members of Stanford SALT Lab for their help and feedback throughout this project.

References

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?, 2025. URL https://arxiv.org/abs/2503.13657.

Zhuang Chen, Jincenzi Wu, Jinfeng Zhou, Bosi Wen, Guanqun Bi, Gongyao Jiang, Yaru Cao, Mengting Hu, Yunghwei Lai, Zexuan Xiong, and Minlie Huang. Tombench: Benchmarking theory of mind in large language models, 2024. URL https://arxiv.org/abs/2402.15052.

Yuyang Cheng, Yumiao Xu, Chaojia Yu, and Yong Zhao. Hawk: A hierarchical workflow framework for multi-agent collaboration, 2025. URL https://arxiv.org/abs/2507.04067.

Tim R. Davidson, Adam Fourney, Saleema Amershi, Robert West, Eric Horvitz, and Ece Kamar. The collaboration gap, 2025. URL https://arxiv.org/abs/2511.02687.

Meta Fundamental AI Research Diplomacy Team (FAIR)†, Anton Bakhtin, Noam Brown, Emily Dinan, Gabriele Farina, Colin Flaherty, Daniel Fried, Andrew Goff, Jonathan Gray, Hengyuan Hu, Athul Paul Jacob, Mojtaba Komeili, Karthik Konath, Minae Kwon, Adam Lerer, Mike Lewis, Alexander H. Miller, Sasha Mitts, Adithya Renduchintala, Stephen Roller, Dirk Rowe, Weiyan Shi, Joe Spisak, Alexander Wei, David Wu, Hugh Zhang, and Markus Zijlstra. Human-level play in the game of diplomacy by combining language models with strategic reasoning. *Science*, 378(6624):1067–1074, 2022. doi: 10.1126/science.ade9097. URL https://www.science.org/doi/abs/10.1126/science.ade9097.

Xianzhe Fan, Xuhui Zhou, Chuyang Jin, Kolby Nottingham, Hao Zhu, and Maarten Sap. Somi-tom: Evaluating multi-perspective theory of mind in embodied social interactions. In *NeurIPS D&B*, 2025. URL https://arxiv.org/abs/2506.23046.

Johannes Forkel and Jakob Foerster. Entropy is all you need for inter-seed cross-play in hanabi, 2025. URL https://arxiv.org/abs/2511.22581.

Adam Fourney, Gagan Bansal, Hussein Mozannar, Cheng Tan, Eduardo Salinas, Erkang, Zhu, Friederike Niedtner, Grace Proebsting, Griffin Bassman, Jack Gerrits, Jacob Alber, Peter Chang, Ricky Loynd, Robert West, Victor Dibia, Ahmed Awadallah, Ece Kamar, Rafah Hosn, and Saleema Amershi. Magentic-one: A generalist multi-agent system for solving complex tasks, 2024a. URL https://arxiv.org/abs/2411.04468.

Adam Fourney, Gagan Bansal, Hussein Mozannar, Cheng Tan, Eduardo Salinas, Erkang, Zhu, Friederike Niedtner, Grace Proebsting, Griffin Bassman, Jack Gerrits, Jacob Alber, Peter Chang, Ricky Loynd, Robert West, Victor Dibia, Ahmed Awadallah, Ece Kamar, Rafah Hosn, and Saleema Amershi. Magentic-one: A generalist multi-agent system for solving complex tasks, 2024b. URL https://arxiv.org/abs/2411.04468.

Juraj Gottweis, Wei-Hung Weng, Alexander Daryin, Tao Tu, Anil Palepu, Petar Sirkovic, Artiom Myaskovsky, Felix Weissenberger, Keran Rong, Ryutaro Tanno, et al. Towards an ai co-scientist. *arXiv preprint arXiv:2502.18864*, 2025.

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.

Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, et al. Metagpt: Meta programming for a multi-agent collaborative framework. In *The Twelfth International Conference on Learning Representations*, 2023.

Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven K. S. Yau, Zijuan Lin, Liyang Zhou, Chenyu Ran, Lingfeng Xiao, Chenglin Wu, and Jürgen Schmidhuber. MetaGPT: Meta programming for a multi-agent collaborative framework. In *International Conference on Learning Representations*, 2024.

Saffron Huang, Bryan Seethor, Esin Durmus, Kunal Handa, Miles McCain, Michael Stern, and Deep Ganguli. How ai is transforming work at anthropic, 2025. URL https://anthropic.com/research/how-ai-is-transforming-work-at-anthropic/.

Nicholas K Humphrey. The social function of intellect. In *Growing points in ethology*, pp. 303–317. Cambridge University Press, 1976.

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? *arXiv preprint arXiv:2310.06770*, 2023.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. SWE-bench: Can language models resolve real-world GitHub issues? In *International Conference on Learning Representations*, 2024.

Hyunwoo Kim, Melanie Sclar, Xuhui Zhou, Ronan Le Bras, Gunhee Kim, Yejin Choi, and Maarten Sap. Fantom: A benchmark for stress-testing machine theory of mind in interactions, 2023. URL https://arxiv.org/abs/2310.15421.

Ji Woong Kim, Juo-Tung Chen, Pascal Hansen, Lucy Xiaoyang Shi, Antony Goldenberg, Samuel Schmidgall, Paul Maria Scheikl, Anton Deguet, Brandon M White, De Ru Tsai, et al. Srt-h: A hierarchical framework for autonomous surgery via language-conditioned imitation learning. *Science robotics*, 10(104):eadt5254, 2025.

Guohao Li, Hasan Abed Al Kader Hammoud, Hani Itani, Dmitrii Khizbullin, and Bernard Ghanem. CAMEL: Communicative agents for "mind" exploration of large language model society. In *Advances in Neural Information Processing Systems*, 2023a.

Huao Li, Yu Chong, Simon Stepputtis, Joseph Campbell, Dana Hughes, Charles Lewis, and Katia Sycara. Theory of mind for multi-agent collaboration via large language models. In Houda Bouamor, Juan Pino, and Kalika Bali (eds.), *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pp. 180–192, Singapore, December 2023b. Association for Computational Linguistics. doi: 10.18653/v1/2023.emnlp-main.13. URL https://aclanthology.org/2023.emnlp-main.13/.

Hanjun Luo, Chiming Ni, Jiaheng Wen, Zhimu Huang, Yiran Wang, Bingduo Liao, Sylvia Chung, Yingbin Jin, Xinfeng Li, Wenyuan Xu, XiaoFeng Wang, and Hanan Salam. Hai-eval: Measuring human-ai synergy in collaborative coding, 2025. URL https://arxiv.org/abs/2512.04111.

Zhao Mandi, Shreeya Jain, and Shuran Song. Roco: Dialectic multi-robot collaboration with large language models, 2023. URL https://arxiv.org/abs/2307.04738.

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.

Melissa Z Pan, Mert Cemri, Lakshya A Agrawal, Shuyi Yang, Bhavya Chopra, Rishabh Tiwari, Kurt Keutzer, Aditya Parameswaran, Kannan Ramchandran, Dan Klein, Joseph E. Gonzalez, Matei Zaharia, and Ion Stoica. Why do multiagent systems fail? In *ICLR 2025 Workshop on Building Trust in Language Models and Applications*, 2025. URL https://openreview.net/forum?id=wM521FqPvI.

Joon Sung Park, Joseph C. O'Brien, Carrie J. Cai, Meredith Ringel Morris, Percy Liang, and Michael S. Bernstein. Generative agents: Interactive simulacra of human behavior, 2023. URL https://arxiv.org/abs/2304.03442.

David Premack and Guy Woodruff. Does the chimpanzee have a theory of mind? *Behavioral and Brain Sciences*, 1(4):515–526, 1978. doi: 10.1017/S0140525X00076512. Publisher: Cambridge University Press.

Goparaju Purna Sudhakar, Ayesha Farooq, and Sanghamitra Patnaik. Soft factors affecting the performance of software development teams. *Team Performance Management: An International Journal*, 17(3/4):187–205, 2011.

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. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics*, 2024.

Neil Rabinowitz, Frank Perbet, Francis Song, Chiyuan Zhang, SM Ali Eslami, and Matthew Botvinick. Machine theory of mind. In *International conference on machine learning*, pp. 4218–4227. PMLR, 2018.

Kiran Ramnath, Kang Zhou, Sheng Guan, Soumya Smruti Mishra, Xuan Qi, Zhengyuan Shen, Shuai Wang, Sangmin Woo, Sullam Jeoung, Yawei Wang, Haozhu Wang, Han Ding, Yuzhe Lu, Zhichao Xu, Yun Zhou, Balasubramaniam Srinivasan, Qiaojing Yan, Yueyan Chen, Haibo Ding, Panpan Xu, and Lin Lee Cheong. A systematic survey of automatic prompt optimization techniques. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pp. 33066–33098.

Association for Computational Linguistics, 2025. doi: 10.18653/v1/2025.emnlp-main.1681. URL http://dx.doi.org/10.18653/v1/2025.emnlp-main.1681.

Eric Raymond. The cathedral and the bazaar. *Knowledge, Technology & Policy*, 12(3):23–49, 1999.

Prateek Sahoo et al. A systematic survey of prompt engineering in large language models: Techniques and applications. *arXiv preprint arXiv:2402.07927*, 2024.

Yijia Shao, Vinay Samuel, Yucheng Jiang, John Yang, and Diyi Yang. Collaborative gym: A framework for enabling and evaluating human-agent collaboration, 2025. URL https://arxiv.org/abs/2412.15701.

Liangcai Su, Zhen Zhang, Guangyu Li, Zhuo Chen, Chenxi Wang, Maojia Song, Xinyu Wang, Kuan Li, Jialong Wu, Xuanzhong Chen, Zile Qiao, Zhongwang Zhang, Huifeng Yin, Shihao Cai, Runnan Fang, Zhengwei Tao, Wenbiao Yin, Chenxiong Qian, Yong Jiang, Pengjun Xie, Fei Huang, and Jingren Zhou. Scaling agents via continual pre-training, 2025. URL https://arxiv.org/abs/2509.13310.

Haochen Sun, Shuwen Zhang, Lujie Niu, Lei Ren, Hao Xu, Hao Fu, Fangkun Zhao, Caixia Yuan, and Xiaojie Wang. Collab-overcooked: Benchmarking and evaluating large language models as collaborative agents. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pp. 4922–4951. Association for Computational Linguistics, 2025. doi: 10.18653/v1/2025.emnlp-main.249. URL http://dx.doi.org/10.18653/v1/2025.emnlp-main.249.

Wannita Takerngsaksiri, Jirat Pasuksmit, Patanamon Thongtanunam, Chakkrit Tantithamthavorn, Ruixiong Zhang, Fan Jiang, Jing Li, Evan Cook, Kun Chen, and Ming Wu. Human-in-the-loop software development agents, 2025. URL https://arxiv.org/abs/2411.12924.

Wei Tao, Yucheng Zhou, Yanlin Wang, Wenqiang Zhang, Hongyu Zhang, and Yu Cheng. MAGIS: LLM-based multi-agent framework for github issue resolution. *arXiv preprint arXiv:2403.17927*, 2024.

Michael Tomasello. *A natural history of human thinking*. Harvard University Press, 2014.

Sachin Waikar. Artists' perspective: How ai enhances creativity and reimagines meaning, Apr 2021. URL https://hai.stanford.edu/news/artists-perspective-how-ai-enhances-creativity-and-reimagines-meaning.

Ruiyi Wang, Haofei Yu, Wenxin Zhang, Zhengyang Qi, Maarten Sap, Yonatan Bisk, Graham Neubig, and Hao Zhu. Sotopia-π: Interactive learning of socially intelligent language agents. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 12912–12940, 2024a.

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 *The Thirteenth International Conference on Learning Representations*, 2024b.

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. In *International Conference on Learning Representations*, 2025.

Jason Wei et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in Neural Information Processing Systems*, 35:24824–24837, 2022.

Anita Williams Woolley, Christopher F Chabris, Alex Pentland, Nada Hashmi, and Thomas W Malone. Evidence for a collective intelligence factor in the performance of human groups. *science*, 330(6004):686–688, 2010.

Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, Ahmed Hassan Awadallah, Ryen W. White, Doug Burger, and Chi Wang. AutoGen: Enabling next-gen LLM applications via multi-agent conversation. *arXiv preprint arXiv:2308.08155*, 2023.

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.

Zhen Xiang, Linzhi Zheng, Yanjie Li, Junyuan Hong, Qinbin Li, Han Xie, Jiawei Zhang, Zidi Xiong, Chulin Xie, Carl Yang, Dawn Song, and Bo Li. Guardagent: Safeguard llm agents by a guard agent via knowledge-enabled reasoning, 2025. URL https://arxiv.org/abs/2406.09187.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, Chujie Zheng, Dayiheng Liu, Fan Zhou, Fei Huang, Feng Hu, Hao Ge, Haoran Wei, Huan Lin, Jialong Tang, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jing Zhou, Jingren Zhou, Junyang Lin, Kai Dang, Keqin Bao, Kexin Yang, Le Yu, Lianghao Deng, Mei Li, Mingfeng Xue, Mingze Li, Pei Zhang, Peng Wang, Qin Zhu, Rui Men, Ruize Gao, Shixuan Liu, Shuang Luo, Tianhao Li, Tianyi Tang, Wenbiao Yin, Xingzhang Ren, Xinyu Wang, Xinyu Zhang, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yinger Zhang, Yu Wan, Yuqiong Liu, Zekun Wang, Zeyu Cui, Zhenru Zhang, Zhipeng Zhou, and Zihan Qiu. Qwen3 technical report, 2025. URL https://arxiv.org/abs/2505.09388.

John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. SWE-agent: Agent-computer interfaces enable automated software engineering. *arXiv preprint arXiv:2405.15793*, 2024.

Haofei Yu, Zhengyang Qi, Yining Zhao, Kolby Nottingham, Keyang Xuan, Bodhisattwa Prasad Majumder, Hao Zhu, Paul Pu Liang, and Jiaxuan You. Sotopia-rl: Reward design for social intelligence. *arXiv preprint arXiv:2508.03905*, 2025.

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.

Ke Zhang, Xiaoning Zhao, Ce Zheng, Jiahong Ning, Dandan Zhu, Wenqi Zhang, Chen Sun, and Toshiharu Sugawara. Tool-roco: An agent-as-tool self-organization large language model benchmark in multi-robot cooperation, 2025a. URL https://arxiv.org/abs/2511.21510.

Wentao Zhang, Liang Zeng, Yuzhen Xiao, Yongcong Li, Ce Cui, Yilei Zhao, Rui Hu, Yang Liu, Yahui Zhou, and Bo An. Agentorchestra: Orchestrating hierarchical multi-agent intelligence with the tool-environment-agent(tea) protocol, 2025b. URL https://arxiv.org/abs/2506.12508.

Boyuan Zheng, Zeyi Liao, Scott Salisbury, Zeyuan Liu, Michael Lin, Qinyuan Zheng, Zifan Wang, Xiang Deng, Dawn Song, Huan Sun, and Yu Su. Webguard: Building a generalizable guardrail for web agents, 2025. URL https://arxiv.org/abs/2507.14293.

Xuhui Zhou, Hao Zhu, Leena Mathur, Ruohong Zhang, Haofei Yu, Zhengyang Qi, Louis-Philippe Morency, Yonatan Bisk, Daniel Fried, Graham Neubig, et al. Sotopia: Interactive evaluation

for social intelligence in language agents. In *The Twelfth International Conference on Learning Representations*, 2024.

Xuhui Zhou, Valerie Chen, Zora Zhiruo Wang, Graham Neubig, Maarten Sap, and Xingyao Wang. Tom-swe: User mental modeling for software engineering agents. *arXiv preprint arXiv:2510.21903*, 2025.

Hao Zhu, Graham Neubig, and Yonatan Bisk. Few-shot language coordination by modeling theory of mind. In *International conference on machine learning*, pp. 12901–12911. PMLR, 2021.

Kunlun Zhu, Hongyi Du, Zhaochen Hong, Xiaocheng Yang, Shuyi Guo, Zhe Wang, Zhenhailong Wang, Cheng Qian, Xiangru Tang, Heng Ji, and Jiaxuan You. Multiagentbench: Evaluating the collaboration and competition of llm agents, 2025. URL https://arxiv.org/abs/2503.01935.

Mingchen Zhuge, Wenyi Wang, Louis Kirsch, Francesco Faccio, Dmitrii Khizbullin, and Jürgen Schmidhuber. Language agents as optimizable graphs, 2024. URL https://arxiv.org/abs/2402.16823.

A. Dataset Details

This section provides detailed statistics on the CooperBench benchmark. Repository selection criteria are described in §2.3.

A.1. Repository Distribution

Table 3 shows the full breakdown of repositories, features, and task pairs.

Table 3 | Distribution of benchmark tasks across source repositories. Feature counts and task pairs are reported as aggregated totals across base commits (PRs) within each repository.

Language	Repository	#PRs	Features (Σ)	Task Pairs (Σ)	License
Python	DSPy	4	23	55	MIT
	LlamaIndex	3	16	39	MIT
	Pillow	3	15	30	MIT-CMU
	Pallets Click	3	27	115	BSD-3
	Pallets Jinja	3	30	135	BSD-3
	HuggingFace Datasets	3	13	26	Apache-2.0
	Outlines	3	22	79	Apache-2.0
	Tiktoken	1	10	45	MIT
	DirtyEquals	1	9	36	MIT
TypeScript	React Hook Form	2	11	25	MIT
Go	Chi Router	3	13	22	MIT
Rust	Typst	3	10	45	Apache-2.0
Total	**12 repositories**	**34**	**199**	**652**	

Note: Each repository contains 1–4 base commits (PRs), each defining an independent feature pool. Task pairs are constructed within each PR as $\binom{n}{2}$ and summed across PRs.

A.2. Feature Complexity

The final CooperBench benchmark comprises 199 individual features grouped into 52 task sets, yielding 652 evaluated feature pairs. Since the objective is to evaluate coordination rather than raw implementation difficulty, features are intentionally designed to be compact and comparable in difficulty to those found in established code-generation benchmarks. This design ensures that multi-agent failures reflect genuine coordination limitations rather than disproportionate feature complexity.

To quantify feature complexity, we characterize the gold patches for each feature along three axes: (i) *code volume*, measured as the total number of lines added and deleted; (ii) *structural footprint*, captured by the number of modified functions and hunks[4]; and (iii) *modification scope*, defined as the number of files affected. Across the benchmark, features exhibit a deliberately compact footprint. On average, a feature comprises 52.3 changed lines and modifies only 1.4 files, confirming that CooperBench isolates coordination challenges rather than the difficulty of single-agent implementation. Table 4 provides detailed statistics for each repository.

Table 4 | Feature Complexity Statistics by Repository

Language	Repository	Avg. Lines	Avg. Functions	Avg. Files	Easy	Medium	Hard
Python	DSPy	70.9	5.6	1.3	$2_{9\%}$	$4_{17\%}$	$17_{74\%}$
	LlamaIndex	16.8	1.8	1.0	$2_{13\%}$	$14_{87\%}$	$0_{0\%}$
	Pillow	38.1	2.7	1.0	$1_{7\%}$	$11_{73\%}$	$3_{20\%}$
	Pallets Click	53.9	5.4	1.6	$0_{0\%}$	$10_{37\%}$	$17_{63\%}$
	Pallets Jinja	67.7	6.2	1.0	$1_{3\%}$	$14_{47\%}$	$15_{50\%}$
	HuggingFace Datasets	15.3	2.3	1.0	$1_{8\%}$	$11_{85\%}$	$1_{8\%}$
	Outlines	44.7	4.1	1.1	$8_{36\%}$	$6_{27\%}$	$8_{36\%}$
	Tiktoken	46.4	4.6	1.0	$0_{0\%}$	$8_{80\%}$	$2_{20\%}$
	DirtyEquals	71.0	4.0	2.0	$0_{0\%}$	$1_{11\%}$	$8_{89\%}$
TypeScript	React Hook Form	49.8	4.6	2.3	$0_{0\%}$	$8_{73\%}$	$3_{27\%}$
Go	Chi Router	80.2	5.7	2.8	$0_{0\%}$	$5_{38\%}$	$8_{62\%}$
Rust	Typst	58.4	1.7	1.1	$0_{0\%}$	$7_{70\%}$	$3_{30\%}$
Overall	**12 Repositories**	**52.3**	**4.4**	**1.4**	$\mathbf{15_{8\%}}$	$\mathbf{99_{50\%}}$	$\mathbf{85_{43\%}}$

Note: Complexity measured as lines changed (added + removed) and structural elements modified in gold patches. Difficulty categories from SWE-Rater-32B: Easy = <15 min fix, Medium = 15 min–1 hour, Hard = 1–4 hours.

B. LLM-based merge conflict resolver

CooperBench evaluates cooperation on merged code. When patch merging produces textual conflicts, we use a small learned resolver to remove conflict markers while preserving both sides' intent. We train a small local resolver rather than calling a larger proprietary model so that the merge step remains narrow and predictable, avoids fixing anything beyond trivial merge cleanup, and can run locally. At evaluation time, we invoke the learned resolver only after a standard merge attempt and a union merge attempt do not yield a test passing merged artifact.

We construct training data by replaying merges between independently produced feature patches and extracting the conflict marked regions from conflicted files. We identify each conflict region by scanning for Git conflict markers <<<<<<<, =======, and >>>>>>>. We extract the marked block together with a small fixed context window, default $c = 5$ lines before and after.

[4]A hunk is a contiguous block of changed lines in a diff, representing a localized code modification.

We generate synthetic conflicts by perturbing these real conflict snippets. Our default generator is `gpt-4o`. This keeps training examples representative of our patch distribution while avoiding direct reuse of repository specific content. For each real or synthetic conflict snippet, we create a reference resolution with `gpt-5` and fine tune a small code model, `Qwen/Qwen2.5-Coder-0.5B-Instruct`, using LoRA based supervised fine tuning (SFT). We train for three epochs with a maximum sequence length of 2048 tokens. When the resolver is invoked, we extract the conflicted region with its fixed context window, run deterministic decoding with temperature $= 0$, and replace that region with the model's resolution. We release the trained resolver as `Qwen2.5-Coder-0.5B-Merge-Resolver`.[5]

C. Difficulty-stratified evaluation

Raw success rates are insufficient for comparing coordination overhead across models. A model dropping from 50% Solo to 30% Coop has the same 20-point gap as one dropping from 80% to 60%, but the first loses 40% of its capability while the second loses only 25%. We need a metric that accounts for baseline differences. We also want to integrate across task difficulty rather than rely on aggregates that mask variation. This section derives such a metric using the relative difficulty defined in Section 4.

We partition tasks into 10 equal-width buckets over the normalized difficulty range $[0, 1]$ and compute success rate at each bucket midpoint, with 95% Wilson confidence intervals that remain well-calibrated near 0 and 1. This produces two curves per model, one for Solo and one for Coop.

We summarize each curve by its area under the curve (AUC) via trapezoidal integration. The absolute gap $\Delta_{\text{AUC}} = \text{AUC}_{\text{Solo}} - \text{AUC}_{\text{Coop}}$ measures coordination cost but depends on baseline. We therefore report *retention* $= \text{AUC}_{\text{Coop}} / \text{AUC}_{\text{Solo}}$, which normalizes for capability. A retention of 0.64 means 64% of Solo performance survives coordination.

For aggregate statistics across models we sum raw counts rather than averaging rates, which preserves proper weighting when models have different sample sizes.

[5] `huggingface.co/CodeConflict/Qwen2.5-Coder-0.5B-Merge-Resolver`

Algorithm 1: Constructing difficulty-stratified success curves

Input: Task set with difficulty scores $d(t) \in [0, 1]$, success outcomes for Solo and Coop per model

Output: Success curves with 95% CIs, AUC gap, and retention per model and pooled

```
// Bucket tasks by difficulty
```
Split $[0, 1]$ into 10 equal buckets;
Assign each task to its bucket based on $d(t)$;

```
// Compute curves per model
```
foreach *model* m **do**
  foreach *bucket* b **do**
    Compute Solo success rate $r_{m,b}^{\text{Solo}} = k_{m,b}^{\text{Solo}} / n_{m,b}$;
    Compute Coop success rate $r_{m,b}^{\text{Coop}} = k_{m,b}^{\text{Coop}} / n_{m,b}$;
    Compute 95% Wilson CI for each rate;
  end
  Compute AUC_{Solo} and AUC_{Coop} via trapezoidal integration;
  Compute $\Delta_{\text{AUC}} = \text{AUC}_{\text{Solo}} - \text{AUC}_{\text{Coop}}$;
  Compute retention $= \text{AUC}_{\text{Coop}} / \text{AUC}_{\text{Solo}}$;
end

```
// Pool across models
```
foreach *bucket* b **do**
  Sum counts across models to get pooled n_b and k_b;
  Compute pooled rates and Wilson CIs;
end
Compute pooled AUC gap and retention;

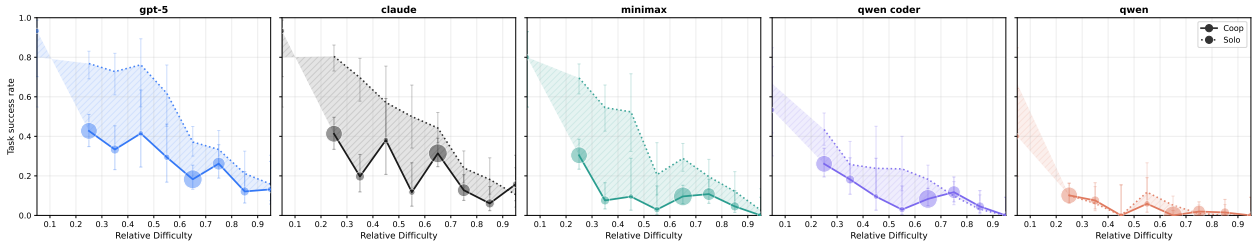


Figure 7 | Success rate versus relative difficulty for Solo and Coop settings. Shaded regions indicate 95% Wilson confidence intervals. The gap between curves represents coordination cost, which is largest at mid-difficulty.

On average, 41% of Solo capability is lost when agents must coordinate (pooled retention 0.59). The pattern across models reinforces that coding ability does not predict coordination ability. `MiniMax` exhibits the worst retention (0.46) despite mid-tier coding performance, while `Qwen` achieves the highest retention (0.68) despite being the weakest coder. Weak models may benefit from a floor effect, but `MiniMax` demonstrates that strong coding provides no protection against coordination overhead.

Table 5 | Coordination retention by model. Retention measures what fraction of Solo AUC is preserved under Coop. Higher values indicate better coordination capability.

Model	Counts (k)		AUC		Derived	
	Solo	Coop	Solo	Coop	Δ_{AUC}	Retention
gpt-5	315	183	0.506	0.325	0.181	0.64
claude	307	168	0.469	0.283	0.186	0.60
minimax	236	91	0.374	0.171	0.203	0.46
qwen coder	141	87	0.236	0.148	0.088	0.63
qwen	41	30	0.106	0.072	0.034	0.68
pooled	1039	558	0.338	0.200	0.138	0.59

D. Prompt Optimization: Failure-Driven Design

This appendix documents the iterative optimization of the collaborative setting execution prompt through systematic failure analysis. Following established prompt engineering practices (Ramnath et al., 2025; Sahoo et al., 2024), we employed an evidence-based approach: beginning with a basic prompt and incrementally adding sections to address specific failure modes observed in agent behavior. **The prompt shown below represents the final, stable version used consistently across all experimental runs reported in this paper.**

Through iterative refinement, we identified three primary failure categories requiring explicit prompt guidance: *context misunderstanding* (agents treating coordination as optional), *spatial coordination failures* (overlapping edits due to vague messages), and *coordination protocol failures* (missing final status updates). The final prompt structure directly maps to these failure categories.

Collaborative Setting Execution Prompt

Role: You are `{{ agent_id }}` working on the following feature in parallel with another agent.

Scenario: You are working on separate branches implementing different features, but your implementations will be tested by 2-way merging both branches to main. You must prevent any merge conflicts.

Feature Description:
`{{ feature_description }}`

Implementation Plan:
`{{ plan }}`

Your Task:

1. Implement the feature according to the plan.
2. You can communicate with the other agent using MCP tools:
   - `openhands_comm_send`: Send messages to the other agent
   - Messages from the other agent will appear automatically as `'[Inter-agent message]'`
3. Coordinate to avoid conflicts by specifying exact file paths and line numbers.
4. Complete the implementation.

Coordination Requirements:

- Share your implementation approach early with specific line ranges so both agents can coordinate.
- If the other agent reports working on the same file, discuss who modifies which specific line ranges to avoid conflicts.
- **Never** use insertion markers or comments like `// [handleSubmit:onFinally] other agent inserts` – these cause merge conflicts.
- Instead, coordinate by dividing the file into non-overlapping sections with specific line ranges.
- Before you stop or complete your work, you **must** send a final status update message to the other agent summarizing what you've implemented.

Merge Conflict Prevention:

- Think of this as two developers working on separate branches that will be merged together.
- Any overlapping changes to the same lines will cause merge conflicts.
- Coordinate line-by-line to ensure no overlap in your modifications.

Work directory: `{{ workspace }}`

Failure-to-Prompt Mapping The scenario section addresses context misunderstanding by explicitly establishing that agents work on separate branches that will be merged, making coordination mandatory. Analysis showed that many agents in early versions did not coordinate until after starting implementation; with the scenario section, most agents coordinate during planning. The coordination requirements section addresses spatial coordination failures through multiple mechanisms. The exact line number requirement (with concrete example) addresses vague coordination messages, significantly reducing spatial conflicts. The insertion marker prohibition substantially reduced marker-related conflicts. The mandatory final status update requirement increased compliance and reduced incomplete handoff failures. The merge conflict prevention section reinforces context understanding through a mental model and technical explanation of merge conflict mechanisms, helping agents understand why coordination matters and how to prevent conflicts.

Design Decisions The prompt follows a specific ordering: (1) *Identity* establishes agent role, (2) *Scenario* sets merge conflict constraints before task description, (3) *Feature* and (4) *Plan* provide context, (5) *Task* describes what to do, (6) *Requirements* specify how to coordinate, and (7) *Prevention* reinforces understanding. This ordering follows the principle that constraints should precede task descriptions (Sahoo et al., 2024). Language choices employ mandatory language for critical behaviors and strong prohibitions for anti-patterns, as optional language was frequently ignored. Concrete examples are included rather than abstract guidance, consistent with findings that concrete examples improve prompt effectiveness (Wei et al., 2022). All experimental results reported in this paper were obtained using this final prompt version.

E. Communication ablation

Section 5 reports that communication does not improve cooperation success. Table 6 provides the full breakdown across merge strategies. We evaluate three merging approaches in sequence: Naive (standard git merge), Union (accept both sides on conflict), and LLM (our learned resolver from App. B). The Δ column shows the net effect of communication on final merge success after all resolution steps. Communication slightly improves Naive merge rates by reducing raw conflicts, but this advantage disappears after Union and LLM resolution. The final effect is near zero or slightly negative across all models.

Table 6 | Merge success (%) on the 652-task summary. Subscripts show Δ from prior column; final column shows comm effect.

	No-comm			With-comm			
Model	Naive	Union	LLM	Naive	Union	LLM	Δ
GPT-5	13.88	26.69$_{+12.8}$	27.91$_{+1.2}$	20.42	26.64$_{+6.2}$	27.90$_{+1.3}$	-0.1
Claude 4.5	12.27	26.84$_{+14.6}$	27.30$_{+0.5}$	16.72	24.85$_{+8.1}$	25.92$_{+1.1}$	-1.4
MiniMax-M2	8.62	14.72$_{+6.1}$	14.88$_{+0.2}$	7.36	11.50$_{+4.1}$	13.96$_{+2.5}$	-0.9
Qwen3-Coder	6.90	12.88$_{+6.0}$	14.72$_{+1.8}$	6.75	12.42$_{+5.7}$	13.34$_{+0.9}$	-1.4
Qwen3-Instruct	1.53	3.22$_{+1.7}$	3.37$_{+0.2}$	2.30	4.45$_{+2.1}$	4.60$_{+0.2}$	+1.2
Avg.	8.64	16.87$_{+8.2}$	17.64$_{+0.8}$	10.71	15.97$_{+5.3}$	17.14$_{+1.2}$	-0.5

F. Communication error detection

We use an LLM-as-judge to classify communication failures for Section 5. Abstract labels like "hallucination" are difficult for LLMs to apply reliably, so we instead define fine-grained categories anchored to quotable evidence. The judge must cite exact quotes from the conversation and omits the label if evidence is weak. We then aggregate these detections into three high-level categories for reporting.

Communication Error Detection Prompt

You are a careful reviewer of two agent collaboration conversations. This is a **precision-first** detector of bad conversation patterns. Prefer returning no issue unless the evidence is strong and explicit.

Important exclusion. Do not label state mismatch or visibility confusion itself as an error (e.g., agents on separate branches unable to see each other's changes). Bad conversation patterns around these topics should still be labeled.

Taxonomy. Label at most one category per conversation.

- **C1a** Unanswered direct question (no reply)
- **C1b** Unanswered direct question (ignored)
- **C2** Non-answer or vague answer
- **C4a** Incorrect claim (uncorrected)
- **C3b** Incorrect claim (corrected)
- **C4a** Spammy repetition (repeats same information)
- **C4b** Spammy repetition (near-duplicate status blocks)

Evidence requirements. Include at least two exact quotes that make the issue undeniable. C1a/C1b require the question plus demonstration of missing or irrelevant response. C3a requires the incorrect claim and later contradiction. C4a/C4b require two quotes showing the repetition.

Output. Return JSON with `evidence` (list of quotes) and optional `issue` (category id and short description). Omit `issue` if evidence is weak.

Taxonomy design. The eight categories decompose three failure modes into verifiable patterns. *Unresponsiveness* (C1a, C1b, C2) covers questions that receive no reply, are ignored, or get vague non-answers. *Hallucination* (C3a, C3b) covers false claims about code state or completion status. We distinguish corrected from uncorrected claims because uncorrected errors propagate to downstream decisions. *Repetition* (C4a, C4b) covers redundant messages that consume budget without adding information.

G. Failure Symptom Annotation Procedure

We followed a six-stage process, similar in spirit to recent work on multi-agent failure analysis (Cemri et al., 2025). (1) Collect multi-agent-system (MAS) traces from Collaborative runs; (2) identify failures from merged artifacts (e.g., failing tests or missing intended behavior), and link them back to the interaction; (3) develop symptom categories by iterative qualitative coding and resolve disagreements to reach inter-annotator agreement on a shared set of definitions; (4) finalize the resulting symptom set; (5) calibrate an LLM-based annotator on the agreed definitions; and (6) apply the annotator to produce symptom annotations at scale.

Each labeled instance is grounded in three artifacts: (i) *conversation evidence* (the coordination dialogue), (ii) *patch/code evidence* (what each agent changed), and (iii) *outcome evidence* (merge reports and test outputs). A key operational distinction in our rubric is between *implementation failures* (an individual agent delivers incomplete/buggy code regardless of coordination) and *coordination failures* (a breakdown that is only apparent when we consider what agents said and assumed under workspace isolation). Concretely, we require explicit conversation evidence to assign a coordination-failure label; if the only evidence is in the code or error trace, we default to implementation-level failure rather than inferring a coordination breakdown. We codified the final symptom definitions as a structured rubric (including verification requirements and common confusions, e.g., when to

treat "unverifiable claims" versus "work overlap"). We then calibrated an LLM-based annotator on this rubric and required it to emit structured labels (a primary symptom plus any secondary symptoms) together with short supporting evidence snippets.

Human validation. To validate the LLM-based annotator, we randomly sampled 50 trajectories and had human experts independently label them using the same rubric. Human labels matched the LLM annotations on 48 of 50 cases (96% agreement). With $n = 50$ and $\hat{p} = 0.96$, the Wilson 95% confidence interval is [86%, 99%], confirming the annotator's reliability.

H. Symptom examples

We provide representative examples for each coordination failure symptom identified in Table 1.

Work overlap

```
A msg: guid regex includes surrounding curly braces ({. . . }), and A warns about overlap.
A msg: WAIT Agent 10! If you add the section header AND my guid type to your branch, that WILL create a merge conflict!
B msg: I'll add the COMPLETE section (lines 72-81) to my branch, which includes both the section header, your guid type, AND my hash_sha256 type.
```

Explanation: Despite explicit boundaries and warnings, B duplicates A's work; the merged artifact keeps B's (incorrect) version.
Outcome: Wrong regex version wins; GUID tests fail (merged guid pattern missing curly braces).

Divergent architecture

```
A msg: Lines 15-66: My global filter registry code is present. . . Line 69: Still has @dataclass decorator.
B msg: Removed dataclass import. . . Converted Prompt from dataclass to regular class with explicit __init__.
B msg: The 2-way merge should combine our changes cleanly.
Merge: clean (0 conflicts), but A's additions are missing from the merged module.
```

Explanation: Despite explicit expectations about merge behavior, B's class rewrite overwrites the region where A added `register_filter/unregister_filter`; the merge reports 0 conflicts but the exported API is missing.
Outcome: `ImportError: cannot import name register_filter from outlines.prompts`.

Repetition

```
Near-duplicate status updates (different strings, same info).
A msg: I have successfully added the url type to outlines/types/__init__.py at lines 72-77:
A msg: Successfully added to outlines/types/__init__.py at lines 72-77
```

Explanation: The same completion/location is restated with minimal new information, consuming turns without adding constraints a partner can act on.
Outcome: Repeated status updates reduce signal-to-noise and can crowd out missing coordination details.

Unresponsiveness

```
A msg: Which approach would you prefer? I want to ensure we don't lose any functionality while resolving this conflict.
B: no later message answers this question in the conversation log.
```

Explanation: The decision is explicitly requested and never resolved, breaking the coordination loop.
Outcome: The team proceeds without an agreed decision; implementation assumptions diverge.

Unverifiable claims

```
A claim: max_resolution is already added at specific line ranges.
B constraint: I cannot verify your intermediate changes (separate branches).
Result: the merge can be clean while one side's asserted change is silently absent.
```

Explanation: The claim is specific but non-checkable; coordination lacks a verification mechanism (e.g., pasted signature, exact diff, or an agreed placeholder).
Outcome: False shared context about code state leads to incompatible downstream edits.

Broken commitment

```
A msg: I'll add BOTH parameters (fallback_processor and max_batch_size) to the constructor signature, BOTH
docstrings, and BOTH initializations.
A msg: ✓Line 26: Added BOTH parameters (fallback_processor and max_batch_size) to constructor signature.
Observed after merge: constructor only has fallback_processor, missing max_batch_size.
```

Explanation: A makes a confident completion claim that is not corrected or verified by B; under partial observability, this creates false shared context.
Outcome: `TypeError: __init__() got an unexpected keyword argument max_batch_size` (tests fail).

Dependency access

```
Conversation evidence: 0 coordination messages were sent (total_messages=0), so neither agent disclosed
import/initialization risks.
Merge traceback excerpt: src/PIL/Image.py:60 executes from . import ImageDraw. Then src/PIL/ImageDraw.py:45 reads
Image.core.
(PIL.Image still initializing ⇒ circular import failure).
```

Explanation: With no communication at all, the merged import graph is never discussed; the first shared integration check happens only at import time and fails deterministically.
Outcome: `AttributeError: partially initialized module PIL.Image has no attribute core (most likely due to a circular import)`.

Placeholder misuse

```
A msg: I'll add a clear comment marker: [Conditional filters overlay insertion point].
A msg: Please insert your logic immediately AFTER the marker. . . without modifying lines above it.
B msg: Given your marker plan, I didn't alter those methods. . . I rely on __post_init__ to overlay filters.
```

Explanation: The agreed integration point (insert-after-marker) is not used; B implements an alternative wiring path, so the merged decorator surface no longer matches the expected call pattern.
Outcome: `TypeError: prompt got an unexpected keyword argument conditional_filters`.

Parameter flow

```
A msg: renamed edit_file to edit_files with multi-file command construction.
B msg: I'm going to continue. . . based on the current state I see (edit_file method).
B code shape: builds a shell command by interpolating filename into a quoted string, assuming it is a single
string.
```

Explanation: Ambiguity about a changing interface leaves one agent implementing against an outdated contract; after merge, a list flows into string-only formatting.
Outcome: `sed: can't read [. . . ]: No such file or directory` (list passed as a literal string).

Timing dependency

```
A msg: Processing Pipeline: load → image.load → EXIF correction (NEW) → B crop (pending) → mode conversion →
return.
B msg: Applied AFTER EXIF correction (A) and BEFORE mode conversion. . . Pipeline (after merge): load → EXIF
correction → center-crop.
Merge: CLEAN (0 textual conflicts); both declare No conflicts expected.
Merged code excerpt: image = image.crop(. . . )
Merged code absence: no ImageOps.exif_transpose(. . . ) call exists in the merged function.
```

Explanation: They agree on the intended order, but fail to ensure the EXIF correction block is actually present at the agreed insertion point after merge.
Outcome: `assert (640, 480) == (480, 640)` (EXIF correction missing).

I. Case Study: Spatial vs. Semantic Coordination

Section 5 shows that communication reduces merge conflicts but does not improve task success. To understand why, we examine a representative failure in detail. This case illustrates the distinction between *spatial coordination* (agreeing on which lines to edit) and *semantic coordination* (agreeing on what values and behaviors to implement). We find that agents excel at the former but neglect the latter.

I.1. Task Setup

The task comes from the Jinja2 template engine. Jinja2 provides a `groupby` filter that groups items by an attribute. For example, `users|groupby("city")` groups a list of users by their city. The original function signature looks like this:

```
def sync_do_groupby(
    environment: "Environment",
    value: "t.Iterable[V]",
    attribute: t.Union[str, int],
    default: t.Optional[t.Any] = None,
) -> "t.List[t.Tuple[t.Any, t.List[V]]]":
```

Two agents must independently extend this filter with new parameters. Agent 1 adds a `case_sensitive` parameter that controls whether grouping treats "CA" and "ca" as the same value. The intended default is `False`, matching the existing `sort` filter's case-insensitive behavior. Agent 2 adds a `reverse` parameter that controls whether groups appear in descending order.

The coordination challenge is that both parameters must be added to the same function signature. Git performs conflict detection at the *hunk* level (contiguous blocks of changes), not individual lines. If both agents modify the signature independently, Git will detect a conflict because both changes touch the same hunk. This creates a genuine coordination requirement where agents cannot simply work in isolation.

I.2. What Agents Communicated

The agents exchanged 10 messages totaling over 3,000 words. Their communication focused almost entirely on spatial coordination: line numbers, file paths, and edit ranges.

Agent 1: I'll be modifying the `sync_do_groupby` and `do_groupby` functions in `src/jinja2/filters.py` **lines 1161–1230**... According to our coordination plan, both parameters will be added together to avoid merge conflicts.

Agent 2: I'll wait for you to complete the implementation in `src/jinja2/filters.py` **lines 1161–1230**...

Agent 1: Your reverse insertion points remain:
`- sync: for _, values in groupby(sorted(value, key=group_expr), group_expr):`
Please proceed to add `reverse: bool = False` immediately after `case_sensitive`.

This communication successfully identified the shared code region and established a protocol for who edits what. What the agents never discussed were the semantic details that matter for correctness. Agent 1 correctly states that `case_sensitive` should default to `False` to match the `sort` filter, but this default value is never explicitly confirmed with Agent 2. Neither agent discusses whether the two parameters interact or whether their implementations are independent.

I.3. What Went Wrong

Because both agents anticipated the need to modify the same function signature, each independently added *both* parameters to avoid a Git conflict. However, without semantic coordination, they made inconsistent choices.

Agent 1's patch added only the `case_sensitive` parameter with the correct default:

```
def sync_do_groupby(
    environment: "Environment",
    value: "t.Iterable[V]",
    attribute: t.Union[str, int],
    default: t.Optional[t.Any] = None,
    case_sensitive: bool = False,  # Correct default
) -> "t.List[_GroupTuple]":
```

Agent 2's patch added *both* parameters (to avoid merge conflicts), but reported the wrong value in communication:

Agent 2's status message:
"Signatures now are: (environment, value, attribute, default=None, case_sensitive=True)"

Agent 2 reported `case_sensitive=True` as the default while the correct value is `False`. This discrepancy was never caught because the conversation focused entirely on *where* edits would happen, not *what values* would be used. Neither agent verified the other's actual implementation; they relied on status messages. The semantic meaning of the default ("should match the `sort` filter") was mentioned by Agent 1 but never confirmed by Agent 2.

For reference, the gold (correct) patches show what each feature should look like. The gold patch for `case_sensitive` adds:

```
    default: t.Optional[t.Any] = None,
    case_sensitive: bool = False,
) -> "t.List[_GroupTuple]":
```

And the gold patch for `reverse` adds:

```
    default: t.Optional[t.Any] = None,
    reverse: bool = False,
) -> "t.List[t.Tuple[t.Any, t.List[V]]]":
```

The correct merged signature would combine both:

```
def sync_do_groupby(
    environment: "Environment",
    value: "t.Iterable[V]",
    attribute: t.Union[str, int],
    default: t.Optional[t.Any] = None,
    case_sensitive: bool = False,
    reverse: bool = False,
) -> "t.List[_GroupTuple]":
```

I.4. What Would Have Worked

For this task to succeed, agents needed to coordinate on three levels. *Spatial coordination* they achieved: "I'm editing lines 1161–1230; please add your parameter after mine." *Structural coordination* they partially achieved: "Both parameters go in the signature; I'll add mine first." *Semantic coordination* was missing entirely.

A single message could have prevented the failure:

Missing coordination:
"I'm implementing case_sensitive with default value False (not True). This matches the sort filter's case-insensitive default. If you need to include this parameter in your patch, please use exactly case_sensitive: bool = False."

I.5. Implications

This case study provides concrete evidence for the spatial-semantic gap discussed in Section 5. Despite 10 messages and over 3,000 words of coordination, the agents never once discussed the actual default value that `case_sensitive` should have. They successfully negotiated *where* to edit but failed to negotiate *what* to implement. A single clarifying message about the intended default value would have prevented the failure entirely.