OPENCOLLAB: A MULTI-AGENT CODING FRAMEWORK WITH PROGRAMMABLE COLLABORATION AND CONTROLLABLE RUNTIME

Anonymous authors
Paper under double-blind review

ABSTRACT

Multi-agent coding systems are designed to tackle complex software engineering tasks through collaboration. However, existing evaluations typically assume configured organizations are followed faithfully, whereas reality differs. This behavioral gap, combined with differences in underlying system components, prevents clear attribution of observed gains. To this end, we introduce OpenCollab, a multi-agent coding framework that provides a unified infrastructure for programmable collaboration and controllable runtime. Specifically, OpenCollab unifies organization design, enforces experimental control on a shared runtime, and tracks execution through fine-grained event streams. On this basis, we define Adherence to quantify whether the declared organization is actually realized. Our experiments reveal that agents collaborate very differently across configurations: changing any single dimension shifts Adherence, from 47.2% to as high as 97.2%. Furthermore, extensive agentic coding benchmarks show that a two-coder workflow built on OpenCollab establishes new SOTA performance compared to the mainstream harnesses such as Mini-SWE-agent, Codex CLI, and Claude Code, showing that a well-designed organization can outperform strong existing harnesses, while OpenCollab's single-agent configuration uses the fewest tokens across all evaluated suites. OpenCollab establishes a unified multi-agent infrastructure for easy programmable collaboration and controlled causal evaluation. Our code is available at `https://anonymous.4open.science/r/OpenCollab-8898`.

1 INTRODUCTION

Autonomous coding agents have achieved remarkable progress, demonstrating promising capabilities to resolve complex issues in real-world software repositories (Jimenez et al., 2024; Wang et al., 2025). As real-world programming tasks grow in complexity, managing vast contexts and multi-stage execution within a single model becomes increasingly challenging. To push the boundaries of performance, the paradigm is shifting toward multi-agent systems (Li et al., 2023; Hong et al., 2024; Qian et al., 2024; Wu et al., 2024; Dong et al., 2024). By assigning specialized organization, such as analysts, developers, and testers, these systems tackle intricate tasks through division of labor and interactive coordination. This design assumes that structured organization improves performance, yet configured collaboration may not actually take place.

Despite their potential, current multi-agent frameworks lack the infrastructure to evaluate actual effectiveness. First, existing systems lack a unified mechanism to explicitly declare an organizational topology. Second, multi-agent evaluations are rarely controlled, as harness differences and run-to-run variance can obscure what produced a performance gap (Zhang et al., 2026; Kaliyev & Maryanskyy, 2026). Third, intermediate collaborative processes suffer from poor observability, since benchmarks report only final outcomes (Kirgis et al., 2026) and logs are written for human debugging, not attribution. Finally, this opacity obscures a fundamental mismatch between declared and realized organizations, where agents frequently work in isolation without actual delegation (Figure 1). Consequently, standard benchmarks cannot determine whether an organization actually helps, because they wouldn't verify whether collaboration took place.

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

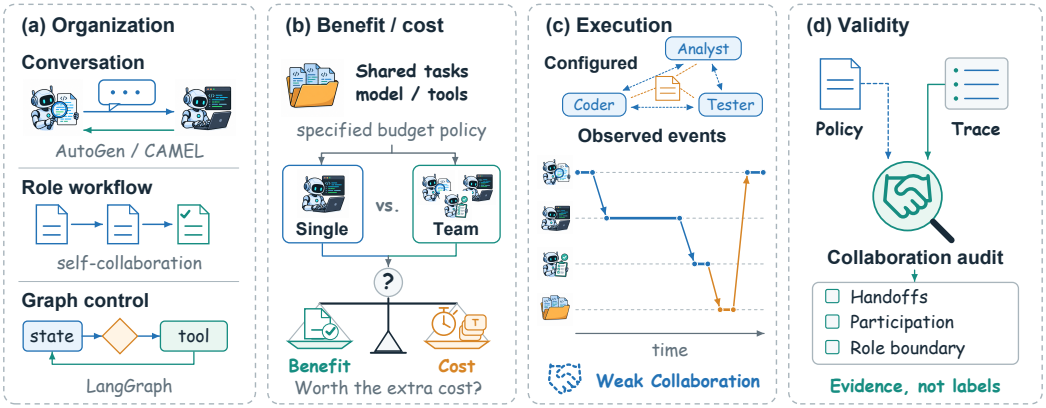

Figure 1: Perspectives on multi-agent collaboration: (a) organization, (b) benefit and cost, (c) configured versus observed execution, and (d) collaboration auditing.

In this paper, we introduce OpenCollab, a multi-agent infrastructure for programmable collaboration and controllable runtime. To enable explicit declaration, OpenCollab provides unified abstractions to statically specify role boundaries and communication topologies. To controlled comparisons, its runtime executes all configurations on a shared substrate, holding non-organizational factors constant. To structured logs, it captures step-by-step actions and resource usage through fine-grained event streams. Building on it, OpenCollab introduces Adherence to quantify the realized organization and, by recording it per run, checks when the complier average causal effect (CACE) (Bloom, 1984; Angrist et al., 1996) can be attributed to the organization.

To systematically evaluate multi-agent systems, we leverage OpenCollab's event streams to dissect their execution across five dimensions (model, tool set, budget, context, and topology). We find that agents collaborate very differently across configurations: changing any single dimension shifts Adherence, from 47.2% to as high as 97.2%; at 47.2%, lead agents monopolize budgets or bypass teammates. Furthermore, our audit of ten mainstream agent frameworks reveals that none natively guarantees controlled experimental comparison or verifies behavioral delivery. Against Claude Code, Codex CLI, and Mini-SWE-agent, OpenCollab establishes new SOTA performance on SWE-bench Pro (64.25%), Terminal-Bench 2.1 (83.15%), and DeepSWE (69.91%). Finally, OpenCollab demonstrates that effective multi-agent collaboration provides substantial gains over working alone. Its collaborative design OpenCollab (Duo) outperforms its single-agent counterpart OpenCollab (Base) by +3.4 points on Terminal-Bench 2.1 and +14.2 points on DeepSWE.

We summarize our main contributions as follows:

- We demonstrate that agents collaborate very differently across configurations: changing a single dimension moves the collaboration rate from 47.2% to as high as 97.2%.
- We formalize Adherence metric to quantify the alignment between a declared organization and its actual runtime execution, grounded in causal identification theory. Furthermore, our rigorous audit of ten mainstream agent frameworks reveals that none can compute Adherence for every organization it runs or ensure controlled comparisons.
- We present OpenCollab, an infrastructure for programmable collaboration and controllable runtime, supporting teams and workflows. It isolates experimental variables and accurately reconstructs execution trajectories via fine-grained event streams.
- On extensive mainstream coding benchmarks, our sample multi-agent OpenCollab (Duo) sets new SOTA performance over other mainstream harnesses, and surpasses the single-agent OpenCollab (Base) by +3.4 points on Terminal-Bench and +14.2 on DeepSWE.

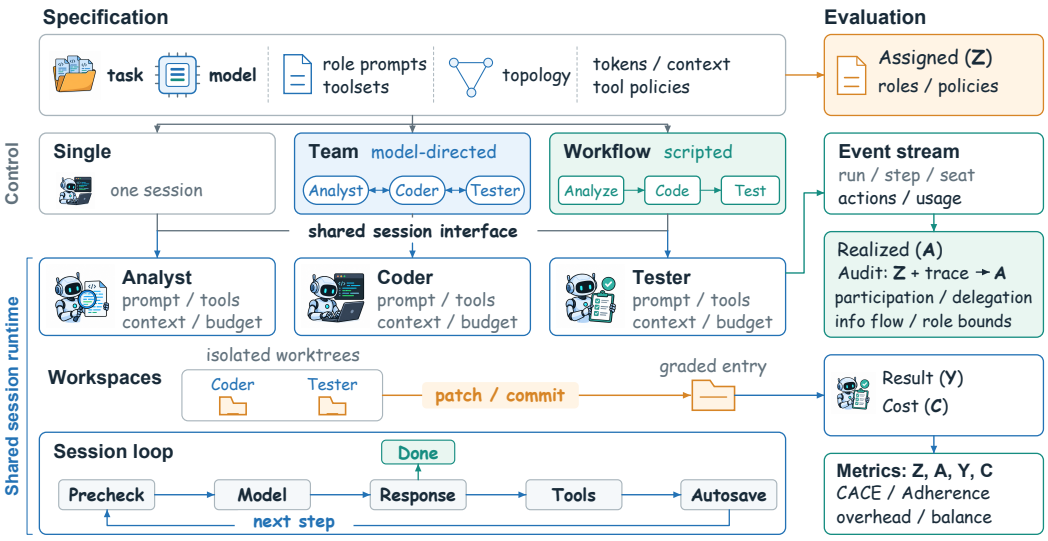

Figure 2: OpenCollab overview. Single, Team, and Workflow controllers share a session runtime. Configuration and execution traces support Adherence and resource costs auditing.

2 Motivation

Deploying multi-agent teams on repository-level tasks typically begins by configuring specialized roles, such as an Analyst, Coder, and Tester (Dong et al., 2024). While intended to divide labor, trajectory analysis reveals that delegation rarely occurs when models decide autonomously. Mostly, the lead agent bypasses configured teammates to resolve issues in isolation, while in other cases, it exhausts its token allowance before issuing any handoff. Consequently, the rate at which the declared organization is realized (formalized as *Adherence*) frequently falls below half as the system collapses into a single-agent loop. We term this divergence between configured and executed organizations the *illusion of collaboration*.

Attempting to resolve this illusion via prompt engineering or log filtering introduces deeper dilemmas. Incrementally mandating delegation in prompts causes Adherence to swing wildly, demonstrating that collaboration is highly sensitive to phrasing rather than organizational structure alone. Conversely, filtering logs to collaborating runs introduces severe post-treatment selection bias. Agents solve easy tasks alone and seek help only when stuck, so this subset skews toward harder tasks. That penalizes the multi-agent arm and invalidates causal claims.

The persistence of this illusion and the failure of simple workarounds stem from underlying architectural limitations in current agent frameworks. First, decentralized resource management allows agents to exhaust budgets locally within isolated execution loops. Because token limits are not governed by a shared pre-call gate across the entire team, lead agents frequently consume the run allowance before reaching a state where delegation could occur. Second, current platforms record execution through conversational prose logs designed for human debugging rather than programmatic analysis. These text transcripts lack role-level attribution, making it difficult to verify which agent executed a specific tool or consumed a given portion of the budget. This opacity prevents automated tools from confirming whether declared topology edges were walked, concealing unexecuted organizations beneath overall benchmark scores.

Addressing these limitations requires three core system capabilities. To prevent unobserved budget exhaustion and eliminate confounders, the framework must operate on a shared runtime enforcing pre-call admission across all arms. To overcome opaque prose logs, execution must be serialized into a structured event stream binding every action and token cost to a distinct role. Finally, to resolve selection bias, the system must compute Adherence from these streams and use CACE to support adherence-aware causal attribution of organizational effects. Together, these capabilities de-

fine an infrastructure that unifies programmable collaboration with controlled evaluation, providing the foundation for OpenCollab.

3 METHODOLOGY

We next describe OpenCollab in detail. In particular, we discuss 1) the architecture of OpenCollab (§3.1), 2) how to program custom collaborative organizations (§3.2), 3) how to ensure fair and controlled comparisons (§3.3), 4) how to trace executions using structured event streams (§3.4), and 5) how to measure collaboration effectiveness with built-in metrics (§3.5). Figure 2 provides an overview of OpenCollab.

3.1 ARCHITECTURE OVERVIEW

OpenCollab introduces a decoupled, two-layer architecture, departing from traditional frameworks where each agent encapsulates its own isolated execution loop. As illustrated in Figure 13 (Appendix C.1), this design separates the high-level orchestration of agent interactions from the low-level mechanics of model execution.

At the foundational layer, OpenCollab operates on a shared session runtime. The fundamental unit of execution is a session, representing a single role in an organization. Every session runs through an identical ten-state finite state machine loop detailed in Appendix C.1. This substrate handles context management, model invocations, and tool execution, ensuring unit consistency across runs.

Built atop the shared runtime, the orchestration layer operates through independent controllers. These controllers govern session topology, lifecycle, and message routing, abstracting control flow away from individual agent implementations. Because all controllers converge on this shared substrate, OpenCollab isolates structural design from low-level runtime mechanics. This decoupling support diverse organizational paradigms without altering the execution engine, removes infrastructure noise, and facilitates both programmable collaboration and controlled evaluation.

3.2 PROGRAMMABLE COLLABORATION

To structure multi-agent execution, the orchestration layer supports two primary collaboration paradigms alongside single-agent baselines, summarized as experimental arms in Table 11 (Appendix C.1): a model-directed team mode and a programmatic workflow mode.

The team controller enables a model-directed collaboration paradigm. Users provide a declarative configuration specifying role prompts, accessible toolsets for each role, and a directed communication topology indicating which roles can address one another, following the declaration schema in Appendix C.3. Once configured, the language model operates within these boundaries, deciding whether, when, and how to delegate subtasks to available teammates. This mode mimics workplace dynamics, allowing multiple agent sessions to cooperate concurrently and pass messages along declared topology edges.

The workflow controller enables programmatic and deterministic agent orchestration. OpenCollab exposes an interface that allows researchers to explicitly govern the collaboration graph. Through simple Python scripts, users can define call orders, fan-out structures, and conditional state transitions across multiple sessions. The script directly controls message routing and termination, ensuring structured execution rather than relying on volatile model decisions.

By separating orchestration from execution, these controllers drastically reduce implementation overhead: reproducing the core multi-agent pattern of *edict* ($\sim$24k lines of code) requires merely 239 lines in OpenCollab. While this enables effortless organizational prototyping, evaluating true collaborative effectiveness still requires strictly controlled conditions.

3.3 CONTROLLED EVALUATION AND COMPARISON

OpenCollab embeds control primitives into the shared runtime to enforce identical conditions across all experimental arms. Specifically, Table 1 defines seven conditions for controlled comparison, comprising five factors held constant at runtime and two structural verification requirements. Ta-

Table 1: Audit of seven controlled evaluation conditions. ✓: by the artifact's own settings and records; –: only through the researcher's code, or in part; ×: not met.

	Held: can the factor be set explicitly?					**Verified: can the run be checked?**	
Artifact	Model	Tools	Budget	Context	Topology	Realized	Compared
Multi Agent frameworks							
AutoGen (Wu et al., 2024)	✓	✓	–	✓	✓	×	×
AG2 (ag2ai, 2024)	–	–	×	–	–	–	×
LangGraph (LangChain, 2023)	–	–	×	–	–	×	×
Coding agents							
SWE-agent (Yang et al., 2024)	✓	✓	–	✓	×	✓	×
OpenHands (Wang et al., 2025)	✓	✓	–	–	–	–	×
Claude Code (Anthropic, 2026)	✓	✓	–	×	–	–	×
Codex CLI (OpenAI, 2025)	✓	–	×	×	–	–	×
DeepSeek Harness (DeepSeek, 2026)	✓	✓	×	–	–	–	×
Evaluation frameworks							
Inspect AI (AI Security Institute, UK, 2024)	✓	✓	–	–	✓	–	–
HAL (Kapoor et al., 2026)	–	×	×	×	×	–	×
OpenCollab (Ours)	✓	✓	✓	✓	✓	✓	✓

ble 7 (Appendix B.1.1) details how the five controlled factors guarantee that non-organizational variables remain aligned across arms, including the base model, per-role tool availability, token and cost budgets, context compaction policies, and the communication topology. The two verification requirements ensure auditability by recording each run's declaration next to its realized trajectory and evaluating one against the other. As our audit of ten artifacts in Appendix C.4 reveals, none of them satisfies all seven conditions, whereas OpenCollab meets all seven.

OpenCollab enforces these conditions through dedicated runtime gates rather than post-hoc filtering. Tool calls and inter-agent messages are intercepted at invocation time, where any request outside the declared role permissions or communication topology is refused and recorded, following the declaration rules in Appendix C.3. For resource control, Figure 13 shows how the runtime enforces an identical per-role token ceiling via pre-call admission within the session state machine. Before each model call, the runtime validates an estimated token reservation against the remaining allowance, halting the session with an explicit termination record if exceeded. Appendix C.5 provides the exact termination codes, which prevent unmetered overruns and cleanly separate runs that exhausted their budget from those that finished voluntarily. These explicit termination records and invocation-time gates form the precise foundation of OpenCollab's execution traceability.

3.4 Execution Traceability and Observability

To capture these records, OpenCollab replaces prose logs with an ordered, append-only event stream. Throughout an execution, every operational step emits a structured record containing a monotonic step index, a timestamp, a run identifier, a role identifier, an event type, and a typed payload. As summarized in Table 13 (Appendix C.5), the runtime defines seven distinct event types to capture the full lifecycle of each role, covering model calls, tool executions, refused actions, session terminations, spawn attempts, worktree modifications, and context compaction triggers.

Because every event is explicitly stamped with its role identifier, the event stream allows automated scripts to reconstruct the realized execution topology and attribute resource costs without parsing natural language text. To ensure physical delivery attribution on the filesystem, non-entry roles in a team operate in isolated Git worktrees over a shared object store (Appendix C.1). Only the entry role workspace is graded upon run completion. Appendix C.5 details this event schema.

3.5 Evaluation of Cooperation Effectiveness

Leveraging the structured event stream, OpenCollab extracts runtime variables to compute automated metrics, addressing the evaluation biases outlined in Section 2. Each run r is represented by the assigned configuration $Z_r \in \{\text{Single}, \text{Workflow}, \text{Team}\}$, the realized-organization indicator $A_r \in \{0, 1\}$, the task result $Y_r \in \{0, 1\}$, and the token cost $C_s(r)$ for each role $s \in \mathcal{S}$, where

$C(r) = \sum_{s \in \mathcal{S}} C_s(r)$ is the total run cost. For Team runs, $A_r = \mathrm{adh}(r)$, the full structural-adherence indicator defined below.

Structural Adherence. Adherence establishes internal validity by verifying that an assigned organizational topology was delivered at runtime. Table 15 (Appendix C.6) defines six verification axes: participation, delegation, role boundary, budget sharing, information flow, and context policy. For each applicable axis x, the runtime produces an objective audit verdict $v_x(r) \in \{\text{adherent}, \text{deviant}, \text{unverifiable}\}$. A run is judged adherent if and only if every applicable axis passes: $\mathrm{adh}(r) = \prod_{x \in \mathcal{X}(Z_r)} \mathbf{1}[v_x(r) = \text{adherent}]$, where $\mathcal{X}(Z_r)$ denotes the set of applicable verification axes for assignment Z_r. The Adherence rate observed across all runs $\mathcal{R}_a$ within an arm, the sample analogue of $\alpha_{\mathrm{adh}} = \Pr(A(z) = 1)$ (Appendix A.1), is:

$$\widehat{\alpha}_{\mathrm{adh}} = \frac{1}{|\mathcal{R}_a|} \sum_{r \in \mathcal{R}_a} \mathrm{adh}(r). \tag{1}$$

A deviant or unverifiable verdict on any applicable axis gives $\mathrm{adh}(r) = 0$, so delegation alone does not make a run adherent: a run in which only one of two required teammates acts has $\mathrm{adh}(r) = 0$. Because $A_r = \mathrm{adh}(r)$ on Team runs, $\widehat{\alpha}_{\mathrm{adh}}$ is the empirical rate at which the assigned Team organization is realized.

Configuration-level complier effect. To resolve the dilution of observed task success gains, OpenCollab divides the Pass@1 difference from Single (the intention-to-treat effect, ITT) by the Adherence rate, giving the complier average causal effect (CACE):

$$\widehat{\mathrm{CACE}} := \widehat{\mathrm{ITT}} \,/\, \widehat{\alpha}_{\mathrm{adh}} = \widehat{\Delta \mathrm{Pass@1}} \,/\, \widehat{\alpha}_{\mathrm{adh}}. \tag{2}$$

Under mean exclusion, $\delta_0 = 0$ (Eq. (8)), the population ratio $\mathrm{ITT}/\alpha_{\mathrm{adh}}$ identifies the configuration-level complier effect: the effect of assigning the target Team configuration rather than Single among runs in which the assigned Team organization is fully realized, not an isolated effect of coordination itself. This causal interpretation relies on two conditions. One-sided compliance holds because the Single arm cannot realize the target Team organization (Appendix A.1). The mean exclusion restriction requires that, among tasks whose Team execution has $A_r = 0$, assigning the Team configuration rather than Single leaves the mean outcome unchanged. Without it, $\mathrm{ITT}/\alpha_{\mathrm{adh}}$ differs from the complier effect by $(1 - \alpha_{\mathrm{adh}})/\alpha_{\mathrm{adh}}$ times that $A_r = 0$ contrast (Theorem A.1, Proposition A.2). While standard evaluations can only assume this restriction, recording A_r per run on paired tasks allows the $A_r = 0$ Team-versus-Single contrast to be estimated, which provides evidence about mean exclusion rather than proof of it (Appendix A.3).

Cost metrics. The same records support two cost metrics, the organizational overhead of runs that do not realize the assigned organization and the workload balance of runs that do; Appendix C.5 defines both, and our experiments do not report them.

4 Experiments

Our evaluations focus on two primary objectives: diagnosing internal collaborative mechanics and measuring system-level effectiveness. We sample tasks from a designated subset of SWE-bench Pro (Deng et al., 2026), Terminal-Bench 2.1 (Merrill et al., 2026; Terminal-Bench, 2026), and DeepSWE (Huang et al., 2026). The cross-harness comparison uses 193, 89, and 113 tasks of these benchmarks, and the adherence experiment uses 36 SWE-bench Pro tasks drawn at random. For cross-harness evaluations, OpenCollab is benchmarked against Claude Code (Anthropic, 2026), Codex CLI (OpenAI, 2025), and Mini-SWE-agent (Yang et al., 2024). We compare three models: DeepSeek-V4.1-Flash, Qwen3.8-Flash, and GPT-5.6-Luna. Extensive experiments and detailed setup can be found in Appendices B.1–B.5.

4.1 Deconstructing Collaboration: Dimensions of Adherence

Standard multi-agent evaluations assume that configured roles operate exactly as declared. We test this by deploying a model-directed team and auditing its execution through OpenCollab's event streams across five configuration dimensions: model, tool set, budget, context, and topology.

Evaluating perturbations against our baseline reference team, Table 2 shows that unconstrained defaults yield only 47.2% Adherence. However, enforcing constraints like restricted tool boundaries

Table 2: Single-dimension ablations relative to the reference team. The reference team uses Qwen3.8-Flash, open prompt cards, unrestricted tools, and 2M tokens per role. Ring topology variants deploy analyst-coder-tester under Mandatory card (Appendix B.1.1). *Unverified* runs are those terminated before collaboration; brackets report 95% Clopper–Pearson confidence intervals.

Dimension	Variant	Adherence (%)	95% CI	Unverified (%)	Pass@1 (%)	Tokens (M)
Model	Qwen3.8-Flash	47.2	[30.4, 64.5]	13.9	75.0	3.05
	DeepSeek-V4.1-Flash	66.7	[49.0, 81.4]	22.2	72.2	3.65
	GPT-5.6-Luna	97.2	[85.5, 99.9]	2.8	61.1	5.79
Tools	All tools	47.2	[30.4, 64.5]	13.9	75.0	3.05
	No edit tools	86.1	[70.5, 95.3]	0.0	66.7	3.24
	Read-only	94.4	[81.3, 99.3]	2.8	52.8	4.24
Budget	0.5M / role	47.2	[30.4, 64.5]	2.8	44.4	0.88
	2M / role	75.0	[57.8, 87.9]	8.3	69.4	4.17
	4M / role	77.8	[60.8, 89.9]	2.8	75.0	5.20
Context	Open card	47.2	[30.4, 64.5]	13.9	75.0	3.05
	Optional card	63.9	[46.2, 79.2]	16.7	61.1	3.58
	Mandatory card	91.7	[77.5, 98.2]	8.3	72.2	4.04
Topology	Star	47.2	[30.4, 64.5]	13.9	75.0	3.05
	Ring + shortcut	66.7	[49.0, 81.4]	13.9	66.7	3.92
	Ring	91.7	[77.5, 98.2]	8.3	63.9	4.21

prevents isolated execution, surging Adherence above 90% and demonstrating that actual collaboration is systematically controllable.

4.2 Causal Validity and Trace Diagnostics

Adherence sets how far CACE rescales the observed ITT. Under the unconstrained Open card, 47.2% Adherence rescales an ITT of $+5.6$ to $+11.8$ (Table 3); this gap follows from low Adherence alone and does not by itself indicate a violation of mean exclusion. Reading $+11.8$ as the configuration-level complier effect further requires the $A_r = 0$ Team-versus-Single contrast to have mean zero (Theorem A.1). The ratio departs from the complier effect by $(1 - \alpha_{\mathrm{adh}})/\alpha_{\mathrm{adh}}$ times this contrast (Proposition A.2), a factor that grows as Adherence falls and is small above 90%. Neither the gap at low Adherence nor the agreement at high Adherence verifies mean exclusion; only the $A_r = 0$

Table 3: Deployment contrasts and adherence-adjusted estimates across configurations on SWE-bench Pro. Each configuration runs the tasks once and is paired task by task with the Single agent. ITT is the Pass@1 difference from Single. Adherence is the share of runs with $A_r = 1$; a run with an unverifiable verdict has $A_r = 0$ (Appendix A.1). CACE is ITT divided by Adherence (Eq. (2)). It has the configuration-level complier interpretation only under the mean-exclusion condition of Appendix A. Brackets span the reported value and the value obtained when unverifiable verdicts are counted as adherent instead; they show sensitivity to that labeling rule and are not a confidence interval. ITT does not depend on adherence and carries no bracket.

Configuration	Adherence (%)	Pass@1 (%)	ITT (%)	CACE (%)
Single	–	69.4	–	–
Open card (reference)	47.2 [47.2, 61.1]	75.0	$+5.6$	$+11.8$ [$+9.1$, $+11.8$]
Mandatory card	91.7 [91.7, 100.0]	72.2	$+2.8$	$+3.0$ [$+2.8$, $+3.0$]
Budget stated, 2M	75.0 [75.0, 83.3]	69.4	0.0	0.0 [0.0, 0.0]
Read-only	94.4 [94.4, 97.2]	52.8	-16.7	-17.6 [-17.6, -17.1]

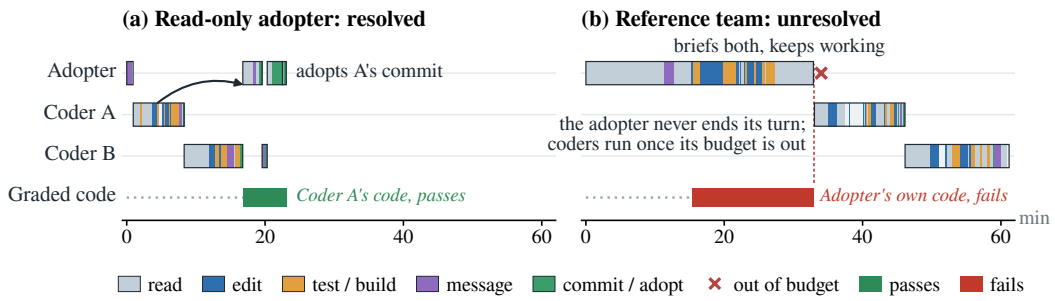

Figure 3: Two execution trajectories from Table 2 on a shared time axis. Lanes denote agents; blocks represent model calls colored by tool usage. (a) Read-only variant: the adopter delegates to Coders and successfully adopts Coder A's patch. (b) Reference team: the adopter briefs Coders but monopolizes execution until exhausting its budget, never submitting teammate work.

contrast bears on it. Because OpenCollab records A_r for every run on tasks also run by Single, this contrast is computed rather than assumed (Appendix A.6). Under the Mandatory card, the paired estimate over adherent runs coincides with CACE in this sample ($+3.0$); under the Open card, the $+5.6$ ITT arises on the 19 tasks with $A_r = 0$ ($+15.8$), whereas the 17 adherent tasks show -5.9, so $+11.8$ should not be read as the complier effect.

OpenCollab's event streams show whether the assigned organization is realized at runtime (Figure 3). In the reference team (Figure 3b), the adopter briefs both coders but never ends its turn, exhausting its token budget while ignoring incoming teammate commits. Restricting the adopter to Read-only tools (Figure 3a) removes its ability to edit files, compelling delegation and clean commit adoption. Further trace analyses are in Appendix B.4.

4.3 System-Level Performance and Organizational Value

Having shown that Adherence responds to configuration, we check that OpenCollab is a competitive harness, so that these findings are not a property of a weak one. Table 4 runs five harnesses with GPT-5.6-Luna on SWE-bench Pro, Terminal-Bench 2.1, and DeepSWE. OpenCollab (Duo) attains the highest Pass@1 on all three (64.25%, 83.15%, and 69.91%), although on SWE-bench Pro its

Table 4: Cross-harness evaluation on SWE-bench Pro, Terminal-Bench 2.1, and DeepSWE.

Benchmark	Harness	Pass@1 (%)	Avg. tokens (M)	Avg. cost ($)	Cache hit (%)
SWE-bench Pro	Mini-SWE-Agent	61.66	4.16	0.90	36.65
	Codex CLI	63.73	7.31	0.73	90.27
	Claude Code	58.03	9.38	2.54	12.50
	OpenCollab (Base)	63.21	3.89	0.41	88.03
	OpenCollab (Duo)	**64.25**	6.50	0.73	84.95
Terminal-Bench 2.1	Mini-SWE-Agent	76.40	2.97	0.45	67.36
	Codex CLI	80.90	2.86	0.26	93.88
	Claude Code	77.53	21.95	2.73	79.32
	OpenCollab (Base)	79.78	1.54	0.19	78.38
	OpenCollab (Duo)	**83.15**	4.60	0.60	76.47
DeepSWE	Mini-SWE-Agent	61.95	19.36	3.27	58.81
	Codex CLI	47.79	15.01	1.30	96.47
	Claude Code	56.64	77.90	7.66	91.16
	OpenCollab (Base)	55.75	11.95	1.19	90.53
	OpenCollab (Duo)	**69.91**	26.11	2.67	89.36

margin over Codex CLI (63.73%) is one task in 193. OpenCollab (Base) uses the fewest tokens and the lowest cost on all three benchmarks, and OpenCollab (Duo) costs less than Claude Code on each (Appendix B.2).

OpenCollab (Duo) is a Workflow in which code issues every handoff, so both of its Coders run on every task. Paired by task against OpenCollab (Base), it solves 21 DeepSWE tasks that the single agent fails and fails 5 that the single agent solves (69.91% vs. 55.75%); on Terminal-Bench 2.1 the split is 6 to 3 (83.15% vs. 79.78%), too few to separate the two. The workflow spends more tokens, since it runs two complete solving processes and selects between their candidates (Appendix B.1.3).

5 Related Work

Autonomous coding agents increasingly adopt multi-agent architectures, drawing on classical organization theory (Fox, 1981; Galbraith, 1974; Malone & Crowston, 1994; Decker & Lesser, 1993; 1995; HORLING & LESSER, 2004). Modern frameworks operationalize these concepts through multi-agent interaction (Li et al., 2023; Hong et al., 2024; Qian et al., 2024; Chen et al., 2024b; Wu et al., 2024), stateful graphs (LangChain, 2023), or shared-state management (Liu et al., 2026a). These frameworks lower the engineering barrier to deploying complex collaborative topologies. However, existing platforms entangle organizational design with low-level execution, forcing agents to manage clients, contexts, and tools independently. OpenCollab breaks this coupling by isolating orchestration from the runtime, providing a unified substrate to evaluate diverse structures under identical system mechanics.

Yet, whether multi-agent collaboration outperforms single-agent baselines remains contested. Early work highlighted the benefits of multi-agent debate (Du et al., 2024), voting ensembles (Li et al., 2024), and repeated sampling (Brown et al., 2024). Conversely, recent studies report mixed or negative returns from adding agents or calls (Smit et al., 2024; Wang et al., 2024; Chen et al., 2024a; Zhao et al., 2026; Bertalanič & Fortuna, 2026; Cemri et al., 2025), even under equal token budgets (Tran & Kiela, 2026; Kim et al., 2026a). Interacting agent teams often incur substantial coordination overhead (Khatua et al., 2026), while reported gains often fall below the replication noise floor (Kaliyev & Maryanskyy, 2026). These conflicting findings are hard to reconcile because comparisons rarely hold budgets, toolsets, or harnesses constant (Zhang et al., 2026). By enforcing pre-call budget gates and uniform execution primitives, OpenCollab ensures observed performance gaps reflect organizational design rather than infrastructural confounders.

Beyond controlling runtime factors, evaluating collaboration requires confirming that the assigned organization actually materialized during execution. Analyzing interaction networks, Destefanis & Aste (2026) show that designated coordinator roles frequently fail to form communication hubs in practice. Related studies examine plan compliance (Liu et al., 2026b), reversible execution traces (Yu et al., 2026), and role separation (Kim et al., 2026b), while Kirgis et al. (2026) emphasize that agent evaluations require log analysis. Yet existing frameworks treat execution as an unverified black box, reporting outcomes without verifying actual communication. OpenCollab bridges this gap via structured event streams and formal Adherence, providing the infrastructure to audit delivery and support causal attribution of organizational gains under explicit identification assumptions.

6 Conclusion

Evaluating multi-agent systems is challenging because existing frameworks entangle organizational design with execution mechanics. To address this, we present OpenCollab, a unified multi-agent coding platform that cleanly decouples design from execution. This decoupling allows researchers to easily construct diverse multi-agent systems in a unified manner on a shared substrate with identical runtime semantics. Furthermore, OpenCollab tracks runtime Adherence to verify that the configured collaboration actually occurs. Our findings also reveal that the agents collaborate very differently across configurations, so evaluations that do not verify collaboration cannot attribute their gains to it. Moreover, OpenCollab provides both the unified programmability to effortlessly build agent organizations and the controlled runtime required to attribute their effects causally under the explicit identification

AI USE STATEMENT

In this work, we used generative AI tools (LLM-based coding assistants) to help implement parts of the OpenCollab codebase and the evaluation scripts, to give feedback on the experimental design, to run experiments and analyze their results, and to draft and edit parts of the manuscript. We also used them to create figures, search and summarize related literature, format references, and improve readability. The authors reviewed all AI-assisted work: code changes were tested and reviewed, reported numbers were checked against the logged run records, and every reference was checked against its source record. We take full responsibility for the final content of this work, including text, claims, and artifacts produced with the aid of generative AI.

REFERENCES

ag2ai. AG2 (formerly AutoGen): The open-source AgentOS. GitHub repository, 2024. URL https://github.com/ag2ai/ag2.

AI Security Institute, UK. Inspect AI: Framework for large language model evaluations, 2024. URL https://zenodo.org/doi/10.5281/zenodo.18434278.

Joshua D. Angrist, Guido W. Imbens, and Donald B. Rubin. Identification of causal effects using instrumental variables. *Journal of the American Statistical Association*, 91(434):444–455, 1996. doi: 10.1080/01621459.1996.10476902.

Anthropic. Overview - Claude Code docs, 2026. URL https://code.claude.com/docs/en/overview.

Blaž Bertalanič and Carolina Fortuna. The ringelmann effect in multi-agent LLM systems: A scaling law for effective team size. *arXiv preprint arXiv:2606.02646*, 2026. URL https://arxiv.org/abs/2606.02646.

Howard S. Bloom. Accounting for no-shows in experimental evaluation designs. *Evaluation Review*, 8(2):225–246, 1984. doi: 10.1177/0193841x8400800205.

Bradley Brown, Jordan Juravsky, Ryan Ehrlich, Ronald Clark, Quoc V. Le, Christopher Ré, and Azalia Mirhoseini. Large language monkeys: Scaling inference compute with repeated sampling. *arXiv preprint arXiv:2407.21787*, 2024. URL https://arxiv.org/abs/2407.21787.

Mert Cemri, Melissa Z Pan, Shuyi Yang, Lakshya A Agrawal, Bhavya Chopra, Rishabh Tiwari, Kurt Keutzer, Aditya Parameswaran, Dan Klein, Kannan Ramchandran, Matei A Zaharia, Joseph Gonzalez, and Ion Stoica. Why do multi-agent LLM systems fail? In *Advances in Neural Information Processing Systems*, volume 38, Main Conference. Curran Associates, Inc., 2025. doi: 10.52202/085713-4082. URL https://proceedings.neurips.cc/paper_files/paper/2025/file/b1041e52d3be19f0a9bc491657488e4a-Paper-Datasets_and_Benchmarks_Track.pdf.

Lingjiao Chen, Jared Davis, Boris Hanin, Peter Bailis, Ion Stoica, Matei Zaharia, and James Zou. Are more LLM calls all you need? towards the scaling properties of compound AI systems. In *Advances in Neural Information Processing Systems*, volume 37, pp. 45767–45790. Curran Associates, Inc., 2024a. doi: 10.52202/079017-1455. URL https://proceedings.neurips.cc/paper_files/paper/2024/file/51173cf34c5faac9796a47dc2fdd3a71-Paper-Conference.pdf.

Weize Chen, Yusheng Su, Jingwei Zuo, Cheng Yang, Chenfei Yuan, Chi-Min Chan, Heyang Yu, Yaxi Lu, Yi-Hsin Hung, Chen Qian, Yujia Qin, Xin Cong, Ruobing Xie, Zhiyuan Liu, Maosong Sun, and Jie Zhou. AgentVerse: Facilitating multi-agent collaboration and exploring emergent behaviors. In *International Conference on Learning Representations*, volume 2024, pp. 20094–20136, 2024b. URL https://proceedings.iclr.cc/paper_files/paper/2024/file/578e65cdee35d00c708d4c64bce32971-Paper-Conference.pdf.

Keith S. Decker and Victor R. Lesser. Quantitative modeling of complex environments. *Intelligent Systems in Accounting, Finance and Management*, 2(4):215–234, 1993. doi: 10.1002/j.1099-1174.1993.tb00044.x.

Keith S. Decker and Victor R. Lesser. Designing a family of coordination algorithms. In *Proceedings of the First International Conference on Multiagent Systems (ICMAS-95)*, pp. 73–80, San Francisco, CA, USA, 1995. AAAI Press. URL https://cdn.aaai.org/ICMAS/1995/ICMAS95-010.pdf.

DeepSeek. DeepSeek Harness: Everything is a plugin. GitHub repository, 2026. URL https://github.com/deepseek-ai/deepseek-harness.

Xiang Deng, Jeff Da, Edwin Pan, Yannis Yiming He, Charles Ide, Kanak Garg, Niklas Lauffer, Andrew Park, Nitin Pasari, Chetan Rane, Karmini Sampath, Maya Krishnan, Srivatsa Kundurthy, Sean Hendryx, Zifan Wang, Vijay Bharadwaj, Jeff Holm, Raja Aluri, Chen Bo Calvin Zhang, Noah Jacobson, Bing Liu, and Brad Kenstler. SWE-bench pro: Can AI agents solve long-horizon software engineering tasks? In *Proceedings of the 43rd International Conference on Machine Learning (ICML)*, 2026. URL https://icml.cc/virtual/2026/poster/61047.

Giuseppe Destefanis and Tomaso Aste. When agents coordinate: Measuring coordination in multi-agent AI coding. *arXiv preprint arXiv:2608.16801*, 2026. URL https://arxiv.org/abs/2608.16801.

Yihong Dong, Xue Jiang, Zhi Jin, and Ge Li. Self-collaboration code generation via ChatGPT. *ACM Transactions on Software Engineering and Methodology*, 33(7):1–38, 2024. doi: 10.1145/3672459.

Yilun Du, Shuang Li, Antonio Torralba, Joshua B. Tenenbaum, and Igor Mordatch. Improving factuality and reasoning in language models through multiagent debate. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 11733–11763. PMLR, 2024. URL https://proceedings.mlr.press/v235/du24e.html.

Mark S. Fox. An organizational view of distributed systems. *IEEE Transactions on Systems, Man, and Cybernetics*, 11(1):70–80, 1981. doi: 10.1109/tsmc.1981.4308580.

Jay R. Galbraith. Organization design: An information processing view. *Interfaces*, 4(3):28–36, 1974. doi: 10.1287/inte.4.3.28.

Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Ceyao Zhang, Jinlin Wang, Zili Wang, Steven 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*, volume 2024, pp. 23247–23275, 2024. URL https://proceedings.iclr.cc/paper_files/paper/2024/file/6507b115562bb0a305f1958ccc87355a-Paper-Conference.pdf.

Bryan HORLING and Victor LESSER. A survey of multi-agent organizational paradigms. *The Knowledge Engineering Review*, 19(4):281–316, 2004. doi: 10.1017/s0269888905000317.

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

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. SWE-bench: Can language models resolve real-world GitHub issues? In *International Conference on Learning Representations*, volume 2024, pp. 54107–54157, 2024. URL https://proceedings.iclr.cc/paper_files/paper/2024/file/edac78c3e300629acfe6cbe9ca88fb84-Paper-Conference.pdf.

Alibek T Kaliyev and Artem Maryanskyy. How much coordination gain is real? a paired noise-floor protocol for multi-agent LLM benchmarks. *arXiv preprint arXiv:2606.20695*, 2026. URL https://arxiv.org/abs/2606.20695.

Sayash Kapoor, Benedikt Stroebl, Peter Kirgis, Nitya Nadgir, Zachary Siegel, Boyi Wei, Tianci Xue, Ziru Chen, Felix Chen, Saiteja Utpala, Franck Ndzomga, Dheeraj Oruganty, Sophie Luskin, Kangheng Liu, Botao Yu, Amit Arora, Dongyoon Hahm, Harsh Trivedi,

Huan Sun, Juyong Lee, Tengjun Jin, Yifan Mai, Yifei Zhou, Yuxuan Zhu, Rishi Bommasani, Daniel Kang, Dawn Song, Peter Henderson, Yu Su, Percy Liang, and Arvind Narayanan. Holistic agent leaderboard: The missing infrastructure for AI agent evaluation. In *International Conference on Learning Representations*, volume 2026, pp. 98778–98849, 2026. URL https://proceedings.iclr.cc/paper_files/paper/2026/file/a0928f924a344aaebbb7f6cd8d56e34c-Paper-Conference.pdf.

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

Yubin Kim, Ken Gu, Chanwoo Park, Chunjong Park, Samuel Schmidgall, A. Ali Heydari, Yao Yan, Zhihan Zhang, Yuchen Zhuang, Yun Liu, Mark Malhotra, Paul Pu Liang, Hae Won Park, Yuzhe Yang, Xuhai Xu, Yilun Du, Shwetak Patel, Tim Althoff, Daniel McDuff, and Xin Liu. Capable language models can outgrow the benefits of collaboration. *Nature Machine Intelligence*, 8(7): 1157–1172, 2026a. doi: 10.1038/s42256-026-01268-y.

Yubin Kim, Chanwoo Park, Taehan Kim, Eugene Park, Samuel Schmidgall, Salman Rahman, Chunjong Park, Cynthia Breazeal, Xin Liu, Hamid Palangi, Hae Won Park, and Daniel McDuff. TeamBench: Evaluating agent coordination under enforced role separation. *arXiv preprint arXiv:2605.07073*, 2026b. URL https://arxiv.org/abs/2605.07073.

Peter Kirgis, Sayash Kapoor, Stephan Rabanser, Nitya Nadgir, Cozmin Ududec, Magda Dubois, JJ Allaire, Conrad Stosz, Marius Hobbhahn, Jacob Steinhardt, and Arvind Narayanan. Log analysis is necessary for credible evaluation of AI agents. *arXiv preprint arXiv:2605.08545*, 2026. URL https://arxiv.org/abs/2605.08545.

LangChain. LangGraph: Build resilient agents. GitHub repository, 2023. URL https://github.com/langchain-ai/langgraph.

Guohao Li, Hasan 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*, volume 36, pp. 51991–52008. Curran Associates, Inc., 2023. doi: 10.52202/075280-2264. URL https://proceedings.neurips.cc/paper_files/paper/2023/file/a3621ee907def47c1b952ade25c67698-Paper-Conference.pdf.

Junyou Li, Qin Zhang, Yangbin Yu, Qiang Fu, and Deheng Ye. More agents is all you need. *Transactions on Machine Learning Research*, 2024. URL https://openreview.net/forum?id=bgzUSZ8aeg.

Mengyang Liu, Taozhi Chen, Zhenhua Xu, Xue Jiang, and Yihong Dong. Multi-agent collaboration with state management. *arXiv preprint arXiv:2605.20563*, 2026a. URL https://arxiv.org/abs/2605.20563.

Shuyang Liu, Saman Dehghan, Jatin Ganhotra, Martin Hirzel, and Reyhaneh Jabbarvand. From plan to action: How well do agents follow the plan? *arXiv preprint arXiv:2604.12147*, 2026b. URL https://arxiv.org/abs/2604.12147.

Thomas W. Malone and Kevin Crowston. The interdisciplinary study of coordination. *ACM Computing Surveys*, 26(1):87–119, 1994. doi: 10.1145/174666.174668.

Mike Merrill, Alexander Shaw, Nicholas Carlini, Boxuan Li, Harsh Raj, Ivan Bercovich, Lin Shi, Jeong Shin, Thomas Walshe, E. Kelly Buchanan, Junhong Shen, Guanghao Ye, Haowei Lin, Jason Poulos, Maoyu Wang, Marianna Nezhurina, Di Lu, Orfeas Menis Mastromichalakis, Zhiwei Xu, Zizhao Chen, Yue Liu, Robert Zhang, Leon Liangyu Chen, Anurag Kashyap, Jan-Lucas Uslu, Jeffrey Li, Jianbo Wu, Minghao Yan, Song Bian, Vedang Sharma, Ke Sun, Steven Dillmann, Akshay Anand, Andrew Lanpouthakoun, Bardia Koopah, Changran Hu, Etash Guha, Gabriel Dreiman, Jiacheng Zhu, Karl Krauth, Li Zhong, Niklas Muennighoff, Robert Amanfu, Shangyin Tan, Shreyas Pimpalgaonkar, Tushar Aggarwal, Xiangning Lin, Xin Lan, Xuandong Zhao, Yiqing Liang, Yuanli Wang, Zilong (Ryan) Wang, Changzhi Zhou, David

Heineman, Hange Liu, Harsh Trivedi, John Yang, Junhong Lin, Manish Shetty, Michael Yang, Nabil Omi, Negin Raoof, Shanda Li, Terry Yue Zhuo, Wuwei Lin, Yiwei Dai, Yuxin Wang, Wenhao Chai, Shang Zhou, Dariush Wahdany, Ziyu She, Jiaming Hu, Zhikang Dong, Yuxuan Zhu, Sasha Cui, Ahson Saiyed, Arinbjörn Kolbeinsson, Jesse Hu, Christopher Rytting, Ryan Marten, Yixin Wang, Jenia Jitsev, Alex Dimakis, Andy Konwinski, and Ludwig Schmidt. Terminal-bench: Benchmarking agents on hard, realistic tasks in command line interfaces. In *International Conference on Learning Representations*, volume 2026, pp. 40903–40986, 2026. URL https://proceedings.iclr.cc/paper_files/paper/2026/file/444a3737adaee10d86ad2ef5f74468e6-Paper-Conference.pdf.

OpenAI. Codex CLI: Lightweight coding agent that runs in your terminal. GitHub repository, 2025. URL https://github.com/openai/codex.

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 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 15174–15186, Bangkok, Thailand, 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.acl-long.810. URL https://aclanthology.org/2024.acl-long.810/.

Andries Petrus Smit, Nathan Grinsztajn, Paul Duckworth, Thomas D Barrett, and Arnu Pretorius. Should we be going MAD? A look at multi-agent debate strategies for LLMs. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 45883–45905. PMLR, 2024. URL https://proceedings.mlr.press/v235/smit24a.html.

Terminal-Bench. Terminal-Bench 2.1, 2026. URL https://www.tbench.ai/news/terminal-bench-2-1.

Dat Tran and Douwe Kiela. Single-agent LLMs outperform multi-agent systems on multi-hop reasoning under equal thinking token budgets. *arXiv preprint arXiv:2604.02460*, 2026. URL https://arxiv.org/abs/2604.02460.

Qineng Wang, Zihao Wang, Ying Su, Hanghang Tong, and Yangqiu Song. Rethinking the bounds of LLM reasoning: Are multi-agent discussions the key? In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 6106–6131, Bangkok, Thailand, 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.acl-long.331. URL https://aclanthology.org/2024.acl-long.331/.

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

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 conversations. In *First Conference on Language Modeling (COLM 2024)*, 2024. URL https://colmweb.org/2024/AcceptedPapers.html.

John Yang, Carlos Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. SWE-agent: Agent-computer interfaces enable automated software engineering. In *Advances in Neural Information Processing Systems*, volume 37, pp. 50528–50652. Curran Associates, Inc., 2024. doi: 10.52202/079017-1601. URL https://proceedings.neurips.cc/paper_files/paper/2024/file/5a7c947568c1b1328ccc5230172e1e7c-Paper-Conference.pdf.

Simon Yu, Derek Chong, Ananjan Nandi, Dilara Soylu, Jiuding Sun, Christopher D Manning, and Weiyan Shi. Shepherd: Enabling programmable meta-agents via reversible agentic execution traces. *arXiv preprint arXiv:2605.10913*, 2026. URL https://arxiv.org/abs/2605.10913.

Yunbei Zhang, Janet Wang, Yingqiang Ge, Weijie Xu, Jihun Hamm, and Chandan K. Reddy. Stop comparing LLM agents without disclosing the harness. *arXiv preprint arXiv:2605.23950*, 2026. URL https://arxiv.org/abs/2605.23950.

Yuxuan Zhao, Sijia Chen, and Ningxin Su. When does multi-agent collaboration help? an entropy perspective. *arXiv preprint arXiv:2602.04234*, 2026. URL https://arxiv.org/abs/2602.04234.

A IDENTIFICATION THEORY FOR THE CACE

Terminology: Section 3.5 and Table 3 call the ratio $\mathrm{ITT}/\alpha_{\mathrm{adh}}$ CACE for brevity. The derivations below keep the two apart: the ratio is the Wald ratio (its sample value the Wald estimate), and CACE denotes the complier effect $\mathbb{E}[\Delta \mid A = 1]$ of Appendix A.1, which the main text calls the configuration-level complier effect.

Section 3.5 reports the Wald ratio $\mathrm{ITT}/\alpha_{\mathrm{adh}}$, which is read as a complier average causal effect (CACE) only under the mean exclusion restriction below. Here α_{adh} is the probability that a target Team run satisfies the full operational structural-adherence rule. Every applicable axis must be explicitly judged adherent. The Single arm cannot realize that target organization, giving one-sided compliance. The ratio identifies the effect of the assigned Team configuration relative to Single among fully adherent Team runs when the non-adherent Team-versus-Single contrast has mean zero. Compatible paired records can estimate that contrast; otherwise the exact discrepancy and a sensitivity bound state what the ratio may miss.

A.1 SETUP AND NOTATION

Using the notation of Section 3.5, write $Z_r \in \{\mathrm{Single}, \mathrm{Workflow}, \mathrm{Team}\}$ for the assigned arm, $A_r \in \{0, 1\}$ for the realized-organization indicator, and retain outcome $Y_r \in \{0, 1\}$, role cost $C_s(r)$, and arm-specific structural adherence $\mathrm{adh}(r)$. Fix a target Team configuration z and write 0 for the Single baseline. For a task and its pre-specified execution randomness, let $Y(z), Y(0) \in \{0, 1\}$ be the configuration-level potential outcomes under the same scoring rule. Their joint law uses a specified coupling of task and run randomness; their marginal means require only comparable arm distributions. We use the same task weights in both arms and assume consistency of observed runs with their assigned potential outcomes, isolated resets, stable model and runtime versions, and complete scoring under a fixed rule. In particular, failures and timeouts are not dropped according to their outcomes.

For a task and its pre-specified execution randomness, let $A(z) \in \{0, 1\}$ denote the result of applying the operational structural-adherence rule of Section 3.5 to the execution that would be generated under target Team configuration z. With $v_x(z)$ denoting the operational axis verdict generated under that configuration, define

$$A(z) := \prod_{x \in \mathcal{X}(z)} \mathbf{1}\{v_x(z) = \text{adherent}\}. \tag{3}$$

For an observed Team run r assigned configuration z, consistency gives the observed label $A_r = A(z) = \mathrm{adh}(r)$ for that run's task and execution randomness. An adherent verdict contributes one; a deviant or unverifiable verdict contributes zero. Thus $A_r = 1$ exactly when every applicable axis is explicitly judged adherent, and $A_r = 0$ otherwise. Every retained Team run has this label, including runs with unverifiable axis verdicts. Write $A := A(z)$ below and define the population rate $\alpha_{\mathrm{adh}} := \mathbb{P}\{A(z) = 1\}$. The finite-sample rate $\widehat{\alpha}_{\mathrm{adh}}$ reported in Section 3.5 is the mean of the labels A_r over the evaluated Team runs, the sample analogue of this population rate; likewise $\widehat{\mathrm{ITT}}$ is the difference of observed arm means, and $\widehat{\tau}_{\mathrm{Wald}} = \widehat{\mathrm{ITT}}/\widehat{\alpha}_{\mathrm{adh}}$ is the sample Wald estimate. Delegation is one component of the structural-adherence criterion. As illustrated in Appendix C.6, a run can delegate to one teammate while another declared participant never acts; such a run has $A_r = 0$ despite delegation. The cost variables do not enter this outcome identification argument.

Define $A(0) = 0$ only as a statement that a Single run cannot realize the *target Team* organization. It does not score Single's adherence to its own applicable axes as zero; Single can be adherent to its own topology. The pair $(A(0), A(z))$ is therefore either $(0, 0)$ or $(0, 1)$. This is the one-sided-compliance, and hence monotonicity, property relevant to the Wald argument (Bloom, 1984; Angrist et al., 1996).

With $\Delta := Y(z) - Y(0)$, define

$$\mathrm{ITT} := \mathbb{E}[\Delta], \tag{4}$$

$$\mathrm{CACE} := \mathbb{E}[\Delta \mid A = 1] \quad (\alpha_{\mathrm{adh}} > 0), \tag{5}$$

$$\delta_0 := \mathbb{E}[\Delta \mid A = 0] \quad (\alpha_{\mathrm{adh}} < 1). \tag{6}$$

We refer to this configuration-level complier effect as CACE: the effect of assigning the target Team configuration rather than Single among runs that satisfy the full operational structural-adherence

rule. It is not, without further intervention assumptions, an isolated effect of coordination itself. The contrast δ_0 concerns the entire non-adherent stratum $A = 0$; non-delegating runs generally form only a subset of this stratum.

A.2 CACE DECOMPOSITION AND WALD IDENTIFICATION

Theorem A.1 (Decomposition and identification of the CACE)**.** *Under the common consistency and stability conditions above, if* $0 < \alpha_{\mathrm{adh}} < 1$*, then*

$$\mathrm{ITT} = \alpha_{\mathrm{adh}}\ \mathrm{CACE} + (1 - \alpha_{\mathrm{adh}})\,\delta_0. \tag{7}$$

The additional mean exclusion restriction is

$$\delta_0 = \mathbb{E}[Y(z) - Y(0) \mid A = 0] = 0 \tag{8}$$

When it holds, the ratio identifies CACE:

$$\mathrm{CACE} = \frac{\mathrm{ITT}}{\alpha_{\mathrm{adh}}}. \tag{9}$$

If $\alpha_{\mathrm{adh}} = 1$*, the same equality holds without a restriction on the empty* $A = 0$ *stratum. If* $\alpha_{\mathrm{adh}} = 0$*, the CACE is undefined.*

Proof. The law of total expectation, applied to Δ, gives

$$\begin{aligned}\mathbb{E}[\Delta] &= \mathbb{P}(A = 1)\mathbb{E}[\Delta \mid A = 1] + \mathbb{P}(A = 0)\mathbb{E}[\Delta \mid A = 0] \\ &= \alpha_{\mathrm{adh}}\ \mathrm{CACE} + (1 - \alpha_{\mathrm{adh}})\,\delta_0.\end{aligned}$$

Substituting Eq. (8) and dividing by $\alpha_{\mathrm{adh}} > 0$ proves Eq. (9). For $\alpha_{\mathrm{adh}} = 1$, $A = 1$ almost surely, so $\mathrm{ITT} = \mathrm{CACE}$ directly. □

The decomposition follows simply by conditioning on $A(z)$ and does not use $A(0) = 0$. The latter condition supports the one-sided-compliance and "complier" interpretation; it does not itself imply the Wald ratio. Mean exclusion is the additional conditional mean restriction on *all* target-Team runs with $A = 0$, including those with unverifiable axis verdicts; it does not require equal outcomes on every such run. Thus Eq. (9) is the one-sided Wald identification result under mean exclusion and the common conditions, not an unconditional algebraic identity. The arm means identify ITT under the common conditions, and the operational Team adherence labels identify α_{adh}; the additional restriction supplies the ratio's CACE interpretation.

A.3 PAIRED DIAGNOSTIC FOR THE EXCLUSION RESTRICTION

The matched-task design can provide a stronger observation regime than the two arm means. If the recorded tuple $(I, A, Y(z), Y(0))$ is a compatible paired observation from the specified joint law, where I is the task, then

$$\mathrm{CACE} = \mathbb{E}[\Delta \mid A = 1], \qquad \delta_0 = \mathbb{E}[\Delta \mid A = 0] \quad \text{when } \mathbb{P}(A = 0) > 0. \tag{10}$$

Hence the $A = 0$ paired contrast directly estimates the contrast relevant to mean exclusion, and the $A = 1$ paired contrast estimates CACE without using exclusion. A finite-sample estimate of the $A = 0$ contrast provides evidence about the restriction rather than establishing exact equality. This diagnostic uses the full structural-adherence label and includes $A = 0$ runs in which some delegation occurs. Merely sharing a task identifier does not fix the cross-arm dependence of stochastic outcomes: the seed or another explicitly specified coupling protocol (including the product coupling that makes the two arms' execution randomness independent conditional on the task) must make the baseline outcome in each tuple compatible with the same task-and-run coupling used in the definitions above. An unpaired comparison of adherent Team outcomes with the unconditional Single mean need not estimate Eq. (5).

A.4 SENSITIVITY TO EXCLUSION VIOLATIONS

Proposition A.2 (Exact discrepancy and sensitivity bounds)**.** *For $0 < \alpha_{\mathrm{adh}} < 1$, without mean exclusion,*

$$\frac{\mathrm{ITT}}{\alpha_{\mathrm{adh}}} - \mathrm{CACE} = \frac{1-\alpha_{\mathrm{adh}}}{\alpha_{\mathrm{adh}}}\,\delta_0. \tag{11}$$

If $|\delta_0| \le D_{\max}$ for a nonnegative scalar $D_{\max}$, distinct from the indicator $A(z)$, the absolute discrepancy is at most $(1-\alpha_{\mathrm{adh}})D_{\max}/\alpha_{\mathrm{adh}}$. For $\epsilon > 0$, this bound is at most ϵ if and only if

$$\alpha_{\mathrm{adh}} \ge \frac{D_{\max}}{D_{\max}+\epsilon}. \tag{12}$$

If instead $\delta_0 \in [\ell_0, u_0]$, a valid sensitivity interval is

$$\mathrm{CACE} \in \left[\frac{\mathrm{ITT} - (1-\alpha_{\mathrm{adh}})u_0}{\alpha_{\mathrm{adh}}}, \frac{\mathrm{ITT} - (1-\alpha_{\mathrm{adh}})\ell_0}{\alpha_{\mathrm{adh}}}\right] \cap [-1, 1]. \tag{13}$$

Proof. Rearrange Eq. (7) to obtain Eq. (11). Taking absolute values gives the bound; solving $(1-\alpha_{\mathrm{adh}})D_{\max}/\alpha_{\mathrm{adh}} \le \epsilon$ gives Eq. (12). Finally, $\mathrm{CACE} = \{\mathrm{ITT} - (1-\alpha_{\mathrm{adh}})\delta_0\}/\alpha_{\mathrm{adh}}$ decreases with δ_0, giving the interval. Binary outcomes constrain the CACE to $[-1, 1]$. □

The discrepancy in Eq. (11) is a population identification discrepancy, not the finite-sample bias of a random ratio. The interval is valid but need not be sharp after all observed outcome constraints are imposed. A non-delegating-only diagnostic does not bound δ_0 unless it also covers the remaining $A = 0$ runs.

A.5 REPORTING REQUIREMENTS

Table 5 separates the deployment effect from the two ways of learning about configuration-level CACE. All rows require the common scoring, consistency, reset, stability, and task-distribution conditions stated above.

Table 5: Observation regimes and requirements for the Section 3.5 effect.

Target	Observable input	Additional condition or diagnostic
$\mathrm{ITT} = \mathbb{E}[Y(z) - Y(0)]$	Task-weighted arm outcomes	Stable, complete execution identifies the deployment contrast.
$\mathrm{CACE} = \mathrm{ITT}\,/\alpha_{\mathrm{adh}}$	Arm outcomes and target-Team labels A_r; $\alpha_{\mathrm{adh}} = \mathbb{P}(A = 1)$	$\alpha_{\mathrm{adh}} > 0$, target-Team one-sided compliance $A(0) = 0$, and mean exclusion $\delta_0 = 0$.
Paired CACE and diagnostic	Compatible $(I, A, Y(z), Y(0))$ tuples	Directly estimate $\mathbb{E}[\Delta \mid A = 1]$ and, when $A = 0$ occurs, the exclusion-relevant contrast $\delta_0 = \mathbb{E}[\Delta \mid A = 0]$.

A report should retain the task, assigned arm and configuration, outcome, per-axis adherence verdicts (including unverifiable), and the seed or coupling protocol used for any pairing. It should state whether the CACE was obtained from the Wald ratio under mean exclusion or directly from compatible pairs. These are statements about the assigned configuration's effect among Team runs that satisfy the full operational structural-adherence rule, not a separate coordination-only estimand.

A.6 EMPIRICAL CHECK ON THE ADHERENCE EXPERIMENT

Table 6 computes the two paired contrasts of Appendix A.3 for the four configurations of Table 3. Each configuration is paired with the Single agent on the same 36 SWE-bench Pro tasks, and A_r is the full structural-adherence label of that table, with failed and unverifiable runs at $A_r = 0$. For this analysis we take the cross-arm coupling of Appendix A.1 to be the product coupling conditional on the task: conditional on the task I, the Team and Single execution randomness are independent draws. The task-matched tuples in Table 6 are interpreted under this specified coupling; they do

Table 6: Wald ratio and paired contrasts against the Single agent on SWE-bench Pro. Paired, $A_r = 1$ is the mean Team-minus-Single difference over the tasks whose Team run is adherent, the direct estimate of Eq. (10) under the product coupling above; Paired, $A_r = 0$ is the same difference over the remaining tasks, the estimate of δ_0. By construction $\widehat{\mathrm{ITT}} = \widehat{\alpha}_{\mathrm{adh}} \cdot (\text{Paired, } A_r{=}1) + (1 - \widehat{\alpha}_{\mathrm{adh}}) \cdot (\text{Paired, } A_r{=}0)$. Differences in percentage points.

Configuration	Adherence (%)	Tasks A_r=1 / 0	ITT	Wald	Paired, A_r=1	Paired, A_r=0
Open card	47.2	17 / 19	$+5.6$	$+11.8$	-5.9	$+15.8$
Budget stated, 2M	75.0	27 / 9	0.0	0.0	-3.7	$+11.1$
Mandatory card	91.7	33 / 3	$+2.8$	$+3.0$	$+3.0$	0.0
Read-only	94.4	34 / 2	-16.7	-17.6	-14.7	-50.0

not rely on a shared execution seed. Under it, a Team run's A_r is independent of the Single run's outcome on the same task.

The gap between the Wald ratio and the paired estimate over adherent runs narrows as Adherence rises: 17.6 points at 47.2%, 3.7 at 75.0%, and at most 2.9 above 90%, as the factor $(1-\alpha_{\mathrm{adh}})/\alpha_{\mathrm{adh}}$ of Proposition A.2 predicts. Under the Mandatory card the three non-adherent tasks show no difference and the two estimates coincide. Under the Open card the $+5.6$ ITT arises on the 19 non-adherent tasks, while the 17 adherent tasks show -5.9; the Wald ratio of $+11.8$ therefore should not be read as the configuration-level CACE of this configuration.

These estimates are descriptive. Re-running the Single agent unchanged flips the outcome of 5 of the 36 tasks (Appendix B.1), and among the 17 adherent Open-card tasks the two arms disagree on one task only, so the sign of -5.9 is not established. The table supports the direction of the Wald–paired gap across Adherence levels, not a collaboration effect of any single configuration.

B Supplementary Experiments

This part expands Section 4: the setup of its three experiments (B.1), every result table (B.2), how runs end and what they cost (B.3), traced runs (B.4), and the limitations (B.5). Appendix A gives the identification theory behind the Wald estimate of Section 3.5, and Appendix C describes the system.

B.1 Experimental Setup

Section 4 reports three experiments on one runtime. The adherence experiment changes one setting of a team at a time and records whether the declared organization happened (Table 2). The cross-harness comparison runs OpenCollab and three external harnesses on three benchmarks (Table 4). The Duo–single comparison runs a team whose handoffs are fixed by code against a single agent on the same tasks. This subsection gives the task sets, the organizations, the shared settings and the metrics, and then the design of each experiment.

Benchmarks and task sets. The cross-harness comparison uses 193 tasks of SWE-bench Pro (Deng et al., 2026), 89 tasks of Terminal-Bench 2.1 (Merrill et al., 2026) and 113 tasks of DeepSWE (Huang et al., 2026). The adherence experiment uses 36 SWE-bench Pro tasks drawn at random, and every configuration runs on the same 36 tasks. The Duo–single comparison uses the task sets of the cross-harness comparison and is paired task by task on Terminal-Bench 2.1 and DeepSWE.

A SWE-bench Pro run is resolved when every fail-to-pass test passes and no pass-to-pass test breaks, as judged by the benchmark's official evaluator. A run whose tests produce no result counts as unresolved. DeepSWE is graded the same way on each task's fail-to-pass and pass-to-pass tests. A Terminal-Bench 2.1 run is graded by the task's own verifier, which checks the files, program output or running service in the candidate's container; it passes when the verifier returns reward 1. When grading itself failed for an infrastructure reason (a download, a dependency, the verifier not starting), the same candidate was graded again, with no new model call. Nine Claude Code runs that did not end were stopped by hand and scored as failed: eight on DeepSWE and `extract-elf` on Terminal-Bench 2.1; their records keep the verifier's missing reward.

Organizations. The *single agent*, OpenCollab (Base) in Table 4, is one agent with six tools (`bash`, `file_read`, `file_write`, `apply_patch`, `grep`, `git_diff`) and no teammate. A team runs under one of two controllers (Appendix C.1). In a *Workflow* code issues every handoff. In a *Team* the model decides whether to brief a teammate, whom, and how much to do itself. *Duo*, OpenCollab (Duo) of Table 4, is a Workflow: two Coders each produce a candidate fix and code selects one (Appendix B.1.3). The adherence experiment runs two Team rosters, named by their topology. In *Star* an *Adopter* receives the task, may brief two Coders, and chooses and submits the final patch. Each Coder works in its own copy of the repository and reports back with a commit. Whether the Coders run is therefore the Adopter's decision, which is what adherence records. *Ring* has an *analyst* that receives the task and submits, a *coder* and a *tester*. Every agent starts from the single agent's configuration and adds its role card. Figure 4 draws Duo and the three Team topologies.

Shared settings. In the adherence experiment every agent runs with reasoning effort at its highest setting and a streamed response, a limit of 200 steps, and the runtime's reminder to edit after repeated reads switched off. Each agent has its own token allowance of 2M unless a Budget row changes it, and the runtime refuses a call that would exceed it (Appendix C.1). A run is stopped after 5,400 s of wall clock with Qwen3.8-Flash and after 7,200 s with DeepSeek-V4.1-Flash and GPT-5.6-Luna. In this experiment the container in which the agents run has no network access, so an agent cannot fetch a fix from outside the repository; only the runtime, outside the container, reaches the model endpoint.

Which runs are kept. A run that ends for a reason the model caused stays in every denominator: it reached its allowance or the wall clock, produced no patch, or failed the tests. Its patch is graded as it stands. A run lost to an infrastructure fault is launched again on the same task at the same code. A run that still has no result counts as unverified for Adherence and as not resolved for Pass@1, so every configuration is read over its 36 runs.

Metrics. A run is *adherent* when it is adherent on every one of the six axes of Table 15 that applies to its arm. In these experiments the runtime holds four of them (role boundary, budget sharing, information flow and context policy), so only participation and delegation vary (Appendix C.6 gives the reasons). A Team run is therefore adherent when every teammate, every agent other than the entry agent, did work: each spent tokens and produced at least one model output. The teammates are the two Coders in Star and the coder and the tester in Ring, so the two rosters count different agents. A run is *unverified* when the entry agent had already begun to delegate but the run was cut off before every teammate had worked, by the entry agent's token allowance, by the wall clock, or by a teammate that stopped without replying, or when it failed after being launched again. Such a run might have collaborated had it not been cut off, so the log cannot settle it. A run in which the entry agent never delegated is not unverified. Unverified runs count as not adherent ($A_r = 0$, Appendix A.1); counting every unverified run as adherent instead shows how much Adherence depends on this rule. Pass@1 is the share of the 36 runs that are resolved, and Tokens is the median per run over all agents. Intervals are Clopper–Pearson 95%. Two configurations are compared task by task on all 36 tasks, with an exact two-sided sign test and no correction for multiple comparisons. Re-running the single agent unchanged flips the outcome of 5 of the 36 tasks, so a difference of a few tasks between two rows is within run-to-run variation.

B.1.1 THE ADHERENCE EXPERIMENT

Table 7 lists every configuration of this experiment with the code used below.

Every row of Table 2 is one configuration run once on each of the 36 tasks, twelve configurations in all. Every row except the two rings changes one setting of a single reference team: Star with Qwen3.8-Flash, the Open card, all tools, and a 2M-token allowance per agent that the card does not mention. The single agent runs the same 36 tasks twice, the second time unchanged, to measure run-to-run variation.

Model. The same team with DeepSeek-V4.1-Flash or GPT-5.6-Luna in every role. GPT-5.6-Luna is served through a different endpoint from the other two models, so its row changes the endpoint as well as the model.

Table 7: The configurations of the adherence experiment. Each row but S′ is a row of Table 2, or the single agent it is compared with in Table 3; S′ reruns S unchanged. Every agent has a 2M-token allowance except in B1 and B3. Each configuration runs the 36 tasks once. S and S′ are the single agent; Star has an Adopter and two Coders, and both rings use the Ring roster (an analyst, a coder and a tester).

Code	Table 2 row	Card	Model	Entry tools	Topology
S	–	–	Qwen3.8-Flash	all	–
S′	–	–	Qwen3.8-Flash	all	–
R	reference	Open	Qwen3.8-Flash	all	Star
M1	Model	Open	DeepSeek-V4.1-Flash	all	Star
M2	Model	Open	GPT-5.6-Luna	all	Star
T1	Tools	Open	Qwen3.8-Flash	no edit	Star
T2	Tools	Open	Qwen3.8-Flash	read-only	Star
B1	Budget	Open + 0.5M stated	Qwen3.8-Flash	all	Star
B2	Budget	Open + 2M stated	Qwen3.8-Flash	all	Star
B3	Budget	Open + 4M stated	Qwen3.8-Flash	all	Star
C1	Context	Optional	Qwen3.8-Flash	all	Star
C2	Context	Mandatory	Qwen3.8-Flash	all	Star
G1	Topology	Mandatory	Qwen3.8-Flash	all	Ring + shortcut
G2	Topology	Mandatory	Qwen3.8-Flash	all	Ring

Tools. *All tools* is the reference. *No edit tools* removes the Adopter's two file-editing tools (file write and apply patch); it keeps the shell. *Read-only* leaves the Adopter only its reading tools and `adopt`, which checks out a Coder's commit into the Adopter's own tree. The Coders' tools are unchanged.

Budget. Each row adds two sentences to the Adopter's card and sets every agent's allowance to the stated value: "Your budget for this run is X tokens, and each Coder's is the same. On these tasks, a single agent working alone used 1,320,000 tokens on average (median 1,470,000)." The 2M row therefore differs from the reference only in those two sentences.

Context: how the three cards are written. Every Adopter card has the same body: the role, the tools, the two Coders and what each is asked for, and how a run ends. The three cards differ only in the closing section, which Table 8 gives in full. The *Open* card leaves every choice to the Adopter and says that neither keeping the work nor handing it over is expected. The *Mandatory* card opens by saying that the division of the work is not the Adopter's to decide, and then gives one procedure: implementation belongs to the Coders, who are asked one at a time, the second after the first has answered, and the Adopter chooses which candidate to deliver. The *Optional* card gives the same procedure without that opening sentence, and adds one sentence that lets the Adopter depart from it and implement the fix itself. The three cards thus form a ladder: no instruction, an instruction the Adopter may set aside, and an instruction it may not. Because the second Coder is briefed after the first has answered, the two candidates are not independent draws. In the Optional row a Coder that stopped without replying did not notify the Adopter, so an Adopter waiting for that reply could end the run; its Adherence and Pass@1 are therefore not strictly comparable with the other rows.

Topology. The three rows differ in who can message whom (Figure 4). *Star* is the reference team: the Adopter and each Coder can message each other, and the two Coders cannot message each other. The two rings use the Ring roster, each agent with the reference's tools. The analyst's card opens with the same two sentences as the Mandatory card and then asks the analyst to leave the implementation to the coder. In the *Ring*, the analyst can message only the coder, the coder only the tester and the tester only the analyst. *Ring + shortcut* adds one edge, from the coder to the analyst, so the coder may report directly and skip the tester.

Table 8: The closing section of the Adopter's card, the only text that differs between the Open, Optional and Mandatory cards; the rest of the card is identical. Text the Optional and Mandatory cards share is in gray; what distinguishes each card is in black.

Card	Closing section
Open	**What is yours to judge.** Nothing above tells you what order to do things in, whom to talk to, or how much of the work to do yourself. Keeping all of it and handing parts of it over are both open to you, and neither one is what you are expected to do. Those are your calls, and you make them the way you would judge any piece of work: what the request actually needs, what is worth another agent's attention, what it costs to describe a piece of it well enough to hand over, and what it costs to carry all of it in one budget and one conversation.
Optional	**What this run asks of you.** Implementation is the Coders'. There are two of them and neither can see what the other is doing, so ask them one at a time: send the first a message with `message_agent` describing the problem and let it work out its own fix, and when its answer is back, write the second its own brief and let it work out its own. What goes into that second brief is yours – what you learned from the first is yours to use or to leave out. Which of the two this run delivers is yours to choose. **You may depart from this and do the implementation yourself; that is your call, and there are runs where it is the better one.** Use `team_status` first if you want to see who is live. Everything else about how you work – what you read, how you diagnose, what you put in each message – is still yours.
Mandatory	**What this run asks of you. This run is not asking you to decide how to divide the work. It is asking you to divide it a particular way, so do that.** Implementation is the Coders'. There are two of them and neither can see what the other is doing, so ask them one at a time: send the first a message with `message_agent` describing the problem and let it work out its own fix, and when its answer is back, write the second its own brief and let it work out its own. What goes into that second brief is yours – what you learned from the first is yours to use or to leave out. Which of the two this run delivers is yours to choose. Use `team_status` first if you want to see who is live. Everything else about how you work – what you read, how you diagnose, what you put in each message – is still yours.

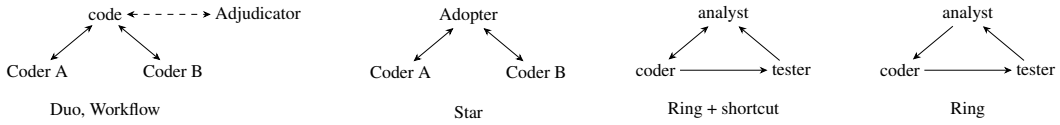

Figure 4: The four organizations. Left: Duo, a Workflow, OpenCollab (Duo) of Table 4; every arrow is a handoff that code makes on every run, and the dashed one runs only when the selection rule does not decide. The other three are the topologies of Table 2, run as Teams: an arrow means the agent at its tail can message the agent at its head, and whether it does is the model's choice.

B.1.2 The cross-harness comparison

This comparison asks whether OpenCollab is a competitive harness, so that what the adherence experiment finds is not a property of a weak one. Five harnesses run on the three benchmarks, all with GPT-5.6-Luna: mini-swe-agent, Codex CLI, Claude Code, OpenCollab (Base) and OpenCollab (Duo).

Network access and answer leakage. On Terminal-Bench 2.1 the runner gives a task internet access when the task's own `allow_internet` flag allows it, and no network otherwise. The trajectories of OpenCollab (Base), OpenCollab (Duo), Claude Code and mini-swe-agent contain successful fetches of external content, so these harnesses did reach the internet on some tasks; for Codex CLI we found no fetch of reference material. Ten generation attempts, on nine harness–task pairs, received material that gives away the answer; all ten are on Terminal-Bench 2.1. We found them by searching the trajectories for paths into the benchmark's repository, `solution` and `tests` directories, reference implementations and reference values, and then re-reading each hit: the tool call, the full text it returned, what the model did next, and whether the resulting candidate was selected. An attempt counts only if the model received the material; a 404, a failed fetch, ordinary project documentation or a search result showing only titles does not count. Most affected

Table 9: OpenCollab (Duo) against OpenCollab (Base), paired by task. p is the exact two-sided sign test on the two middle columns.

Benchmark	Both pass	Only Duo	Only Base	Both fail	p
Terminal-Bench 2.1 (89)	68	6	3	12	0.51
DeepSWE (113)	58	21	5	29	0.0025

tasks were run again from the original task statement and base image in a fresh session, with the old candidate, the old session and any grading feedback kept out, and with a filter on the source and content of what a tool returns; ordinary dependency downloads stayed allowed, so this is not a rerun with the network off. Table 4 reports the results after this handling: every pass obtained with leaked material was withdrawn or replaced.

B.1.3 The Duo–single comparison

The adherence experiment shows that a Team's handoffs may not happen, so a difference in Pass@1 between a Team and a single agent cannot be read as the effect of the organization. OpenCollab (Duo) is a Workflow: code issues every handoff, so both of its Coders run on every task.

The workflow. OpenCollab (Duo) has two solving agents and a judge that is called only when needed. *Coder A* looks for the narrowest root cause and writes a minimal complete fix. *Coder B* works from the same original task and aims at completeness across components, APIs, lifecycles and boundary conditions; it may receive the public test commands that Coder A ran. Each works in its own candidate environment, with the single agent's configuration and six tools. The code then selects between the two candidates by a fixed rule; when the rule does not decide, it calls a read-only *Adjudicator*, which compares the candidates' evidence with the public requirements of the task (on Terminal-Bench 2.1 it also has a read-only tool that pages through that evidence). The code, not an agent, applies the selected candidate.

Paired results. Table 9 pairs OpenCollab (Duo) with OpenCollab (Base) task by task, with an exact two-sided sign test on the tasks where only one of them passed. The difference is significant on DeepSWE and not on Terminal-Bench 2.1.

B.2 Full Results

Table 10 gives, for every configuration of Table 7, the rates behind Table 2 and the task-by-task comparisons behind its reading. A comparison a:b counts the tasks on which only this configuration succeeded (a) and those on which only the other one did (b), over the 36 tasks, with the exact two-sided sign test in parentheses. The reference is R for every Star row and G1 for G2. C1 is not compared, for the reason given under Context in Appendix B.1.1. The single-agent column is left empty for M1 and M2, whose models have no single-agent run.

B.3 How Runs End and What They Cost

This section covers the Qwen3.8-Flash configurations of Table 7, that is, every configuration except M1 and M2. A run ends in one of three ways (Figure 5). It completes when the entry agent (the Adopter, or the analyst in a Ring) submits. It is stopped for its token budget when the entry agent has used up its own allowance before submitting. It is stopped for wall clock when it reaches the 5,400 s limit. Budget stops dominate only where the allowance is smallest: in B1 92% of runs stop for budget and 8% complete. With the largest allowance, B3, a third of the runs reach the wall clock instead. T2 completes most often, in 89% of its runs.

Figures 6 and 7 show what a run costs in tokens and in time, and how a team run's cost splits across its three agents. A single-agent run uses 1.3M to 1.4M tokens on average against an allowance of 2M. A team run uses 2.9M to 4.2M against 6M, except in the two Budget rows whose allowance differs: B1 uses 0.9M of 1.5M and B3 4.7M of 12M. The entry agent takes 33% to 50% of a team's tokens, except in T2, where it can only read and adopt a commit and takes 17%. Because exactly one

Table 10: Adherence and Pass@1 of every configuration, with paired comparisons (only this configuration succeeded : only the other did, exact sign test p). Adherence for a Star run counts both Coders; for a Ring run, the coder and the tester.

	Adherence			Pass@1		
Code	**% [95% CI]**	**Unverified %**	**vs reference**	**%**	**vs reference**	**vs S**
S	–	–	–	69.4	–	–
S′	–	–	–	66.7	2:3 (1.00)	–
R	47.2 [30.4, 64.5]	13.9	–	75.0	–	3:1 (0.63)
M1	66.7 [49.0, 81.4]	22.2	13:6 (0.17)	72.2	2:3 (1.00)	–
M2	97.2 [85.5, 99.9]	2.8	19:1 (<0.001)	61.1	2:7 (0.18)	–
T1	86.1 [70.5, 95.3]	0.0	15:1 (<0.001)	66.7	1:4 (0.38)	1:2 (1.00)
T2	94.4 [81.3, 99.3]	2.8	18:1 (<0.001)	52.8	0:8 (0.008)	2:8 (0.11)
B1	47.2 [30.4, 64.5]	2.8	9:9 (1.00)	44.4	0:11 (0.001)	0:9 (0.004)
B2	75.0 [57.8, 87.9]	8.3	14:4 (0.03)	69.4	1:3 (0.63)	3:3 (1.00)
B3	77.8 [60.8, 89.9]	2.8	13:2 (0.007)	75.0	1:1 (1.00)	3:1 (0.63)
C1	63.9 [46.2, 79.2]	16.7	–	61.1	–	–
C2	91.7 [77.5, 98.2]	8.3	19:3 (<0.001)	72.2	1:2 (1.00)	1:0 (1.00)
G1	66.7 [49.0, 81.4]	13.9	–	66.7	–	2:3 (1.00)
G2	91.7 [77.5, 98.2]	8.3	10:1 (0.01)	63.9	3:4 (1.00)	4:6 (0.75)

agent runs at any moment (Appendix B.5), a run's wall clock divides into the time each agent spends in its own model calls and tool calls, plus the harness's time between turns, which stays below 2%. The time split follows the token split: the entry agent's share ranges from 27% in T2 to 59% in B1. A single-agent run takes 28 to 30 minutes on average and a team run 44 to 59 minutes, except in B1 (20 minutes).

Figure 8 shows when each agent spends its tokens. Each run is cut into ten equal slices of time, from its start to its last model call, and the tokens in each slice are divided among the agents. The entry agent spends nearly all tokens in the first slices, the other two agents take over in the middle, and in every configuration the entry agent's share rises again in the last slice. In B1 the entry agent spends all the tokens of the first three slices; in T2 it spends almost nothing in the middle of the run.

B.4 Three Runs, Traced

Figures 9–11 draw three runs of the adopter–two-coder team from their event streams. Each lane is one agent; an outlined bar is one turn, and a block inside it is one model call, coloured by what the call did. Only one agent runs at a time, so a message to another agent is drawn as a vertical arrow at the moment it is sent, and the dotted line after it is the time the woken agent waits before its turn starts. Labelled points mark moments checked against the trajectory, and the bottom row of each panel is the code that would be graded if the run stopped at that moment; the legend sits above Figure 9, and all three share one page. Each is a single run chosen to show one way the declared organization is or is not realized, so none of them estimates how often that happens, and their time axes differ.

Delegation that works (Figure 9). In the Read-only variant the adopter can read files and adopt a commit, but cannot edit. It sends the same task to both coders. Coder A writes the fix, and every line of the graded patch comes from that edit; Coder B independently writes an equivalent one. The adopter adopts Coder A's commit and asks Coder B to review it, and Coder B agrees.

Briefed but not waited for (Figure 10). In the reference team the adopter briefs both coders at minute 12.7. The tool that sends the brief replies that the teammate runs on its next turn and that ending the turn is how the adopter waits. The adopter instead implements the change itself; at minute 27.2 it writes that it will wait for the coders' replies, then makes five more model calls. Its budget runs out at minute 33 before it submits, and only then do the coders run. Coder A reports its commit to the adopter, which has already stopped. The five target tests the graded code fails reference a

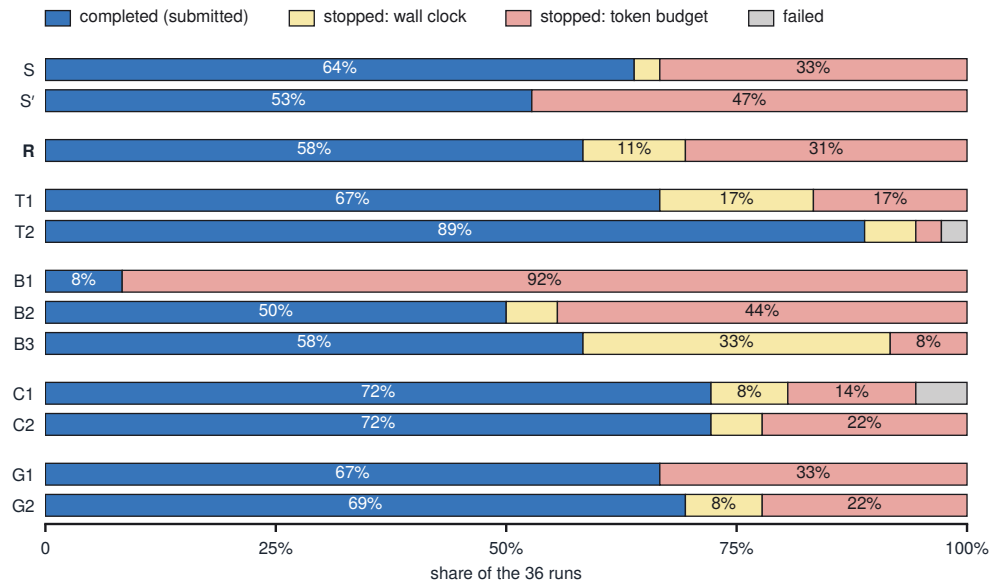

Figure 5: How runs end, as a share of each configuration's 36 runs (Appendix B.1). Completed: the entry agent submitted. Token budget: the entry agent used up its allowance before submitting. Wall clock: the run reached 5,400 s. Failed: no result after the run was launched again. Configuration codes as in Table 7.

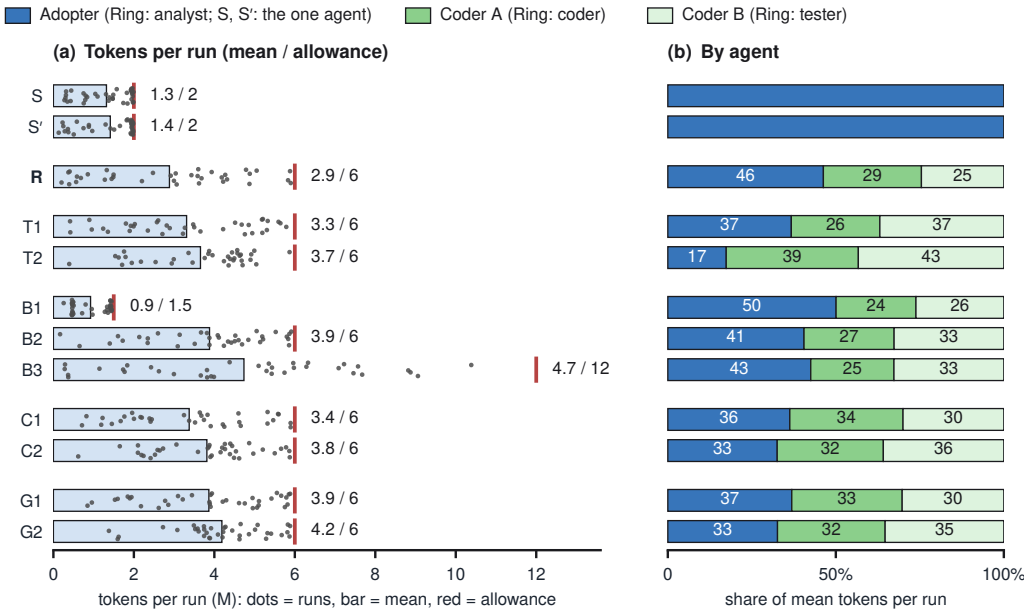

Figure 6: Tokens per run. (a) Each dot is a run; the bar is the mean and the red tick is the run's allowance (each agent's allowance times the number of agents), with "mean / allowance" in millions at the right. (b) The mean split across agents, in percent. In a Ring the three agents are the analyst, the coder and the tester.

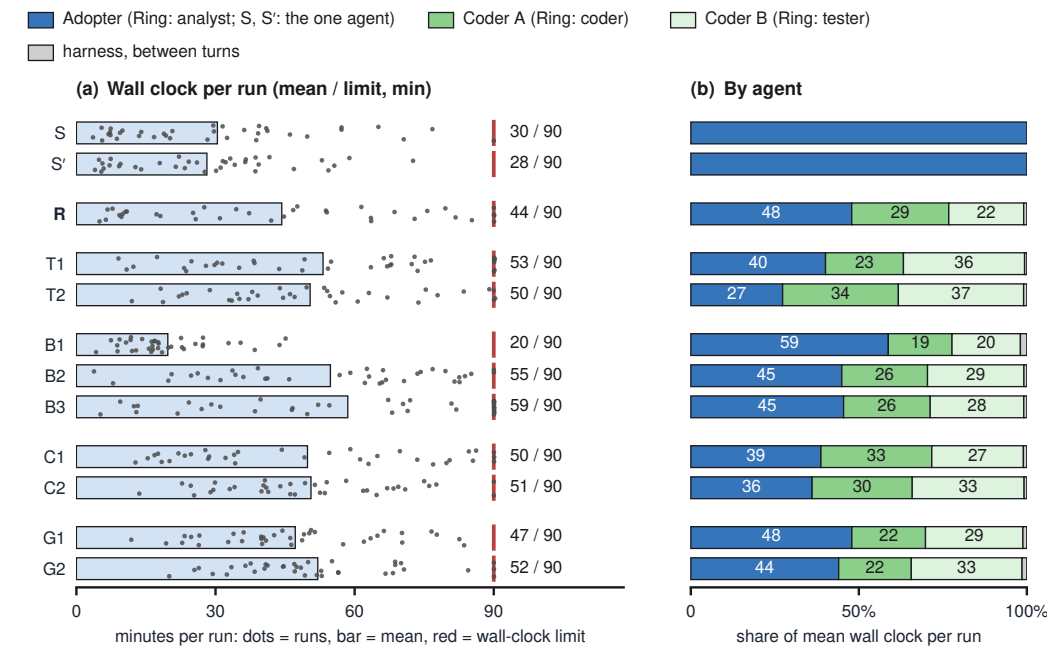

Figure 7: Wall clock per run. (a) Each dot is a run; the bar is the mean and the red tick is the 5,400 s limit, with "mean / limit" in minutes at the right. (b) The mean split across agents, where an agent's time is the duration of its own model calls and tool calls; grey is the harness's time between turns, a thin sliver at the right of each bar.

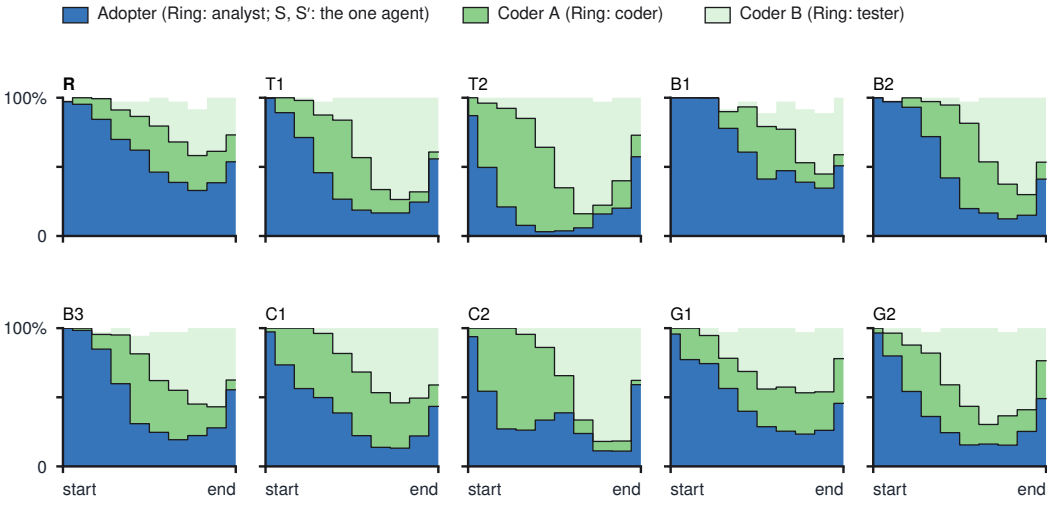

Figure 8: Share of tokens by agent in each tenth of a run, from its start to its last model call, averaged over the configuration's runs. One panel per team configuration of Table 7 with Qwen3.8-Flash.

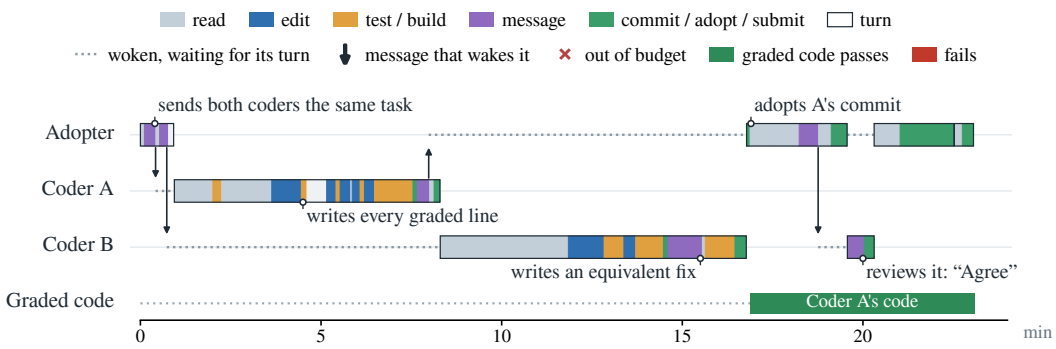

Figure 9: Read-only variant (the adopter can only read files and adopt a commit), resolved: two coders fix the task and the adopter adopts Coder A's commit. A SWE-bench Pro task from the teleport repository.

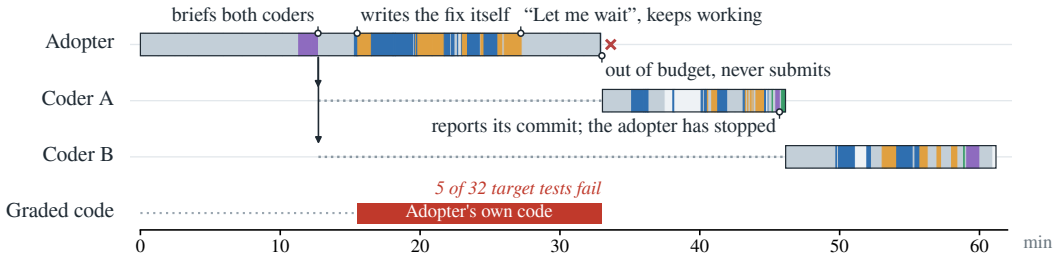

Figure 10: Reference team, unresolved: the adopter briefs both coders but keeps working until its budget ends; the coders run only after its turn ends. A SWE-bench Pro task from the flipt repository.

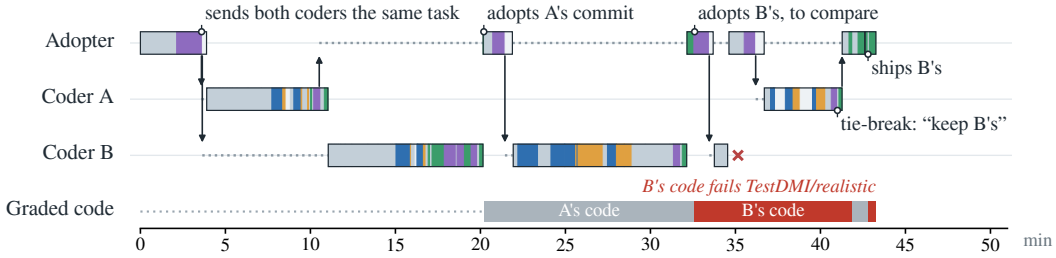

Figure 11: Read-only variant, unresolved: cross-review ships the commit that fails. A SWE-bench Pro task from the teleport repository.

constant name that the task statement does not give, so this failure does not follow from the missing delegation.

Cross-review that picks the failing commit (Figure 11). Again in the Read-only variant, the adopter adopts Coder A's commit, then Coder B's to compare them. Coder A reports that Coder B's commit passes all fifteen of its probes and recommends keeping it, and the adopter ships it. Coder B's code reads a file in a way that bypasses the permission error that a target test injects, and fails that test.

The figures show forms the organization takes; they are not a comparison with the single agent.

B.5 LIMITATIONS

Each limitation below narrows how far a number in this paper reaches.

Three models, not a model sweep. The three model snapshots of Table 2 show that Adherence moves when the model changes, but three snapshots cannot estimate how it varies across models in general.

One agent at a time. All agents in a team share one lock, so exactly one agent runs at any moment; a message to a busy agent is delivered when its turn ends. Organizations whose value depends on agents working truly in parallel are therefore outside what these experiments measure.

One workflow, not the space of workflows. OpenCollab (Duo) is a single decomposition of the task. A different script could deliver a different organization just as faithfully and perform differently.

C System Details

This part describes the code behind Sections 3.1–3.5: the runtime and its two controllers (C.1), how the code is organized (C.2), how a collaboration is declared (C.3), an audit of ten agent artifacts, with OpenCollab scored by the same rule, against the conditions of Section 3.3 (C.4), the run record (C.5), and how adherence is computed from it (C.6).

C.1 Runtime and Control Regimes

Sections 3.1 and 3.2 describe one runtime that carries three control regimes: a Team, a script-ordered Workflow, and a single agent. This appendix gives the limits every session runs under, its context compaction, the two controllers above it, the workspaces, and the token budget.

The session loop. Every session, whichever controller opened it, runs the same loop, a state machine with ten states. `IDLE` precedes the first step. `PRECHECK` admits or refuses the next model call, `CALLING_LLM` makes it, and `HANDLING_RESPONSE` sends a reply with tool calls to `EXECUTING_TOOLS`; `AUTOSAVING` records the step and returns to `PRECHECK`. A tool whose result is deferred parks the session in `AWAITING_EVENTS` until the controller resumes it. `DONE`, `STOPPED` and `ERROR` are the end states (Table 14). A transition that is not in the table of legal transitions raises an error instead of being taken (Figure 12).

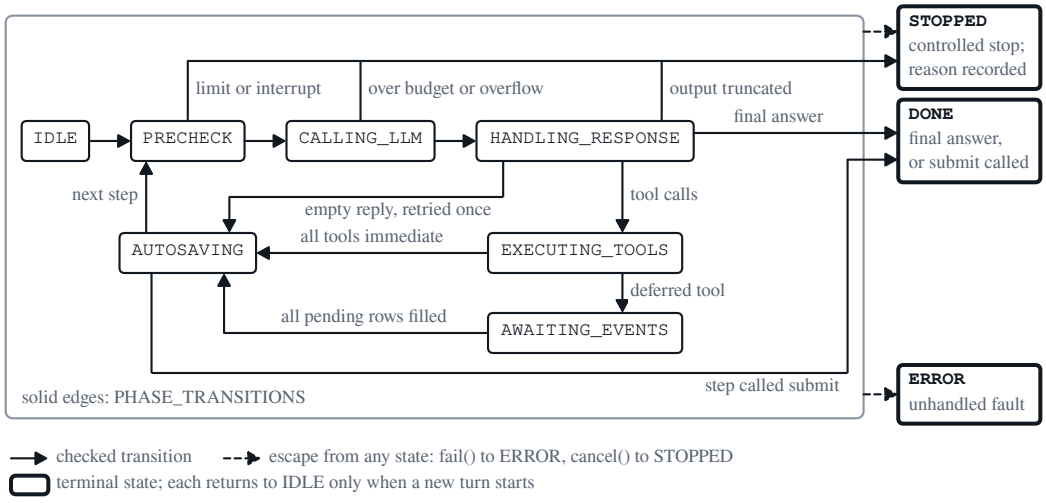

Figure 12: The session state machine that every session runs. Solid edges are the checked transition table; a transition not in the table raises an error instead of being taken. Two escapes bypass the table from any state: an unhandled fault ends the session in `ERROR`, and a cancellation from outside the loop ends it in `STOPPED`. `STOPPED` is a controlled stop at a limit or a refusal, with the reason recorded (Table 14), not a fault. Each end state returns to `IDLE` only when a new turn starts.

Limits. Every limit is checked before a call is made. `PRECHECK` stops the session once it has reached its step limit, its limit on repeated tool calls, or its token allowance. In `CALLING_LLM` the call is then priced with a conservative estimate of its input, and refused if it would not fit in what remains of the allowance. One check also runs after the call, because a call can still cost more than its estimate; a role that ends up past its allowance in this way stops at once. A role stopped at any of these points ends in `STOPPED` with the reason recorded (Table 14). Because an organization decides only which sessions exist and when they start, never what happens inside one, these limits bind identically whichever controller opened the session.

Context compaction. Before each model call the session compacts a copy of its history in four stages applied in order. The first caps every tool result at 16,000 characters and runs on every call. The other three act only when the estimated history exceeds a trigger set 33,000 tokens below the

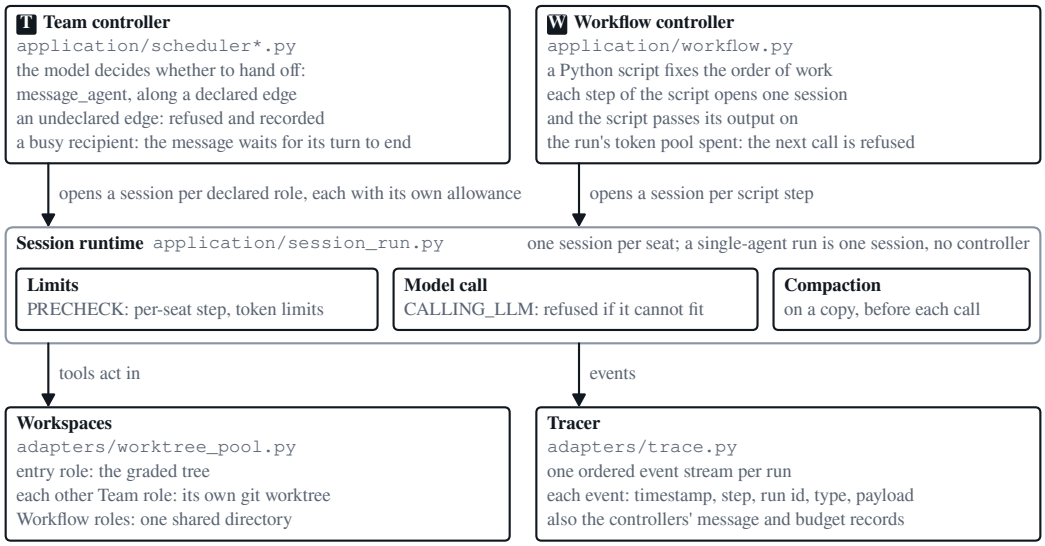

Figure 13: Two controllers over one session runtime. The Team controller opens a session for every declared role, each with its own token allowance; the model decides whether to hand work on with `message_agent`, and a message along an undeclared edge is refused and recorded. The Workflow controller is a Python script that fixes the order of work and opens one session per step; a single-agent run is one session with no controller above it. All sessions run the same code, which checks every limit before a model call, acts in the role's workspace, and writes to one ordered event stream per run (Appendix C.5).

Table 11: The three arms. The last column says how a run is known to have realized its arm. Only in the Team is it a measurement of the model's choice. In the Workflow the script, not the model, issues every handoff it reaches.

Arm	Definition	Differs from Single in	Realized
Single	one agent works the task to completion	—	trivially
Workflow	shared role cards; a script decides the order of work, what each role receives, and when to stop	organization, enforced	by code
Team	shared role cards; the model decides whether to delegate, what to send, and when to stop	organization, offered	measured, $\widehat{\alpha}_{\mathrm{adh}}$

model's context window. They clear the content of old tool results while keeping the five most recent verbatim, then drop whole old tool exchanges, oldest first, and last replace the remaining old span with a summary written by a separate model call. Each stage that fires writes a record to the event log. If the provider still rejects the prompt as too long, a forced pass ignores the trigger and the call is retried once; a second rejection stops the session. The stored history is never altered by compaction.

Two controllers. Above the loop, a controller decides which sessions exist, when each one starts, and how their outputs travel (Figure 13). The Team controller opens a session for every declared role and leaves it to the model whether to send work along the declared edges. The Workflow controller is a Python script that opens sessions in a fixed order and passes their outputs on itself. A single-agent run is one session on the same runtime. Table 11 lists the three arms built from these regimes.

Workspaces. The entry role is the role that receives the task. Its workspace is what the benchmark grades when the run ends, and we call that workspace the graded tree. The Team controller builds every declared teammate before the first model call, in its own git worktree over the same object store, and gives it no task: a teammate starts working only when a message reaches it.

The token budget. Each role has its own fixed token allowance, set before the run. Building a teammate takes nothing from the team's tokens, so a teammate that is never used spends nothing. Grading runs after the run has ended and is not charged to any role's allowance.

C.2 How the Code Is Organized

The system is two code bases. The runtime, the Python package `opencollab`, runs the sessions and the two controllers of Appendix C.1. The evaluator, the package `opencollab_eval`, defines the arms, drives the batches and prepares each run's patch for grading. Every path and count below is taken from the code of the adherence experiment.

Four layers in the runtime. Of the runtime's 167 Python files, 158 fall into four layers (Table 12), and apart from five import statements in the command-line interface, described below, no module imports from a layer above its own. The domain layer holds the session state machine and the other plain data types, and it uses nothing outside the Python standard library. The application layer holds the session loop with its limits, the two controllers and context compaction. The adapters layer holds the code that reaches outside the process: model providers, tools, workspaces and git worktrees, and the event log. The bootstrap layer reads a team file, resolves its tool names to implementations, and assembles the other three layers. Above the layers sits a public surface of nine files: the client, whose `agent`, `team` and `workflow` calls start the three regimes, and the modules for tools, environments, team files and workflows. Figure 14 draws the four layers as the rings of a clean architecture; the layer contract in the code carries that name.

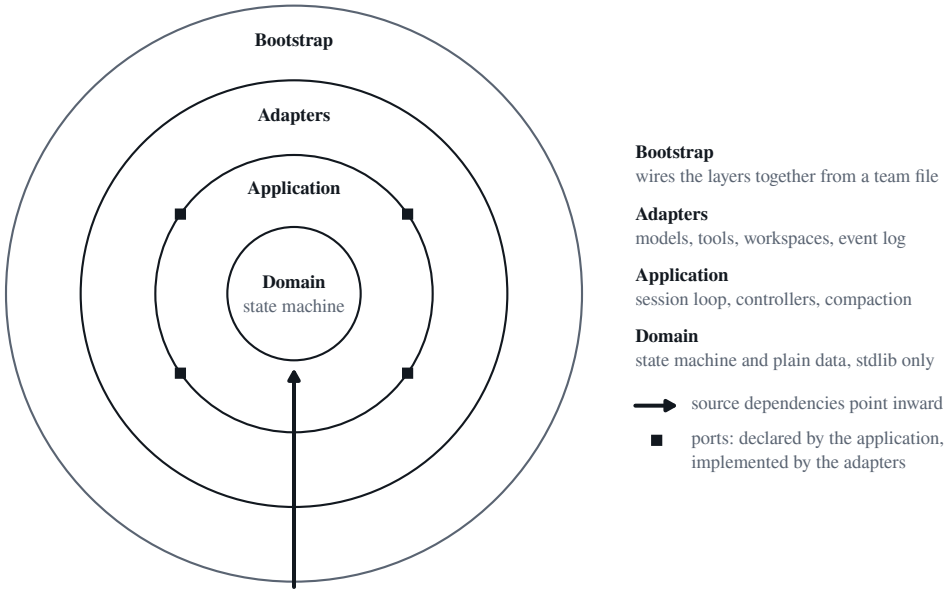

Figure 14: The runtime's four layers as a clean architecture. Every import points inward, from bootstrap to adapters to application to domain; the application declares ports, interfaces that the adapters implement, so it never imports a model provider, a tool or a workspace directly. Figure 15 gives the files in each layer.

Figure 15 draws the same layers with the modules this appendix names.

The layering is checked. Four import contracts, run in continuous integration, state the rules: the order of the layers; that the domain imports none of the runtime's third-party dependencies; that the client and the modules for tools, environments and workflows never import the adapters or the domain directly; and that no two modules of the application, adapters or domain layer import each other in a cycle, except one known cycle between two modules of the Workflow controller. All four hold. We confirmed this by running the contracts on the code of the experiments, and confirmed that the check can fail by adding two forbidden imports to the domain, which broke exactly the two

Table 12: The runtime's layers, top to bottom. Apart from five statements in the command-line interface (see text), no module imports from a layer above its own. *Lines* counts every line of the layer's Python files.

Layer	Files	Lines	What it holds
Public surface	9	894	The client (`sdk/client.py`); the modules for tools, environments, team files and workflows
Bootstrap	22	6,690	Loading a team file (`team_config.py`, Appendix C.3); resolving tool names (`tool_registry.py`); assembling a session, a team or a workflow run
Adapters	72	16,171	Model providers and the per-model capability table (`llm/`); the tools, among them the patch tool, the shell and `message_agent` (`tools/`); local, container and worktree workspaces; the sandbox policy (`safety.py`); the per-run event stream (`trace.py`, Appendix C.5)
Application	51	15,202	The session loop and its limits (`session_run.py`); the Team controller (`application/scheduler.py` and its `scheduler_*` and `_scheduler_*` modules); the Workflow controller (`workflow.py` and its `workflow_*` modules); context compaction (`shaping/`)
Domain	13	1,820	The ten-state session machine and its legal transitions (`session.py`, Appendix C.1); the team topology (`team.py`); the table of sessions that the Team controller keeps, as plain data (`domain/scheduler.py`); agents, events and token estimates

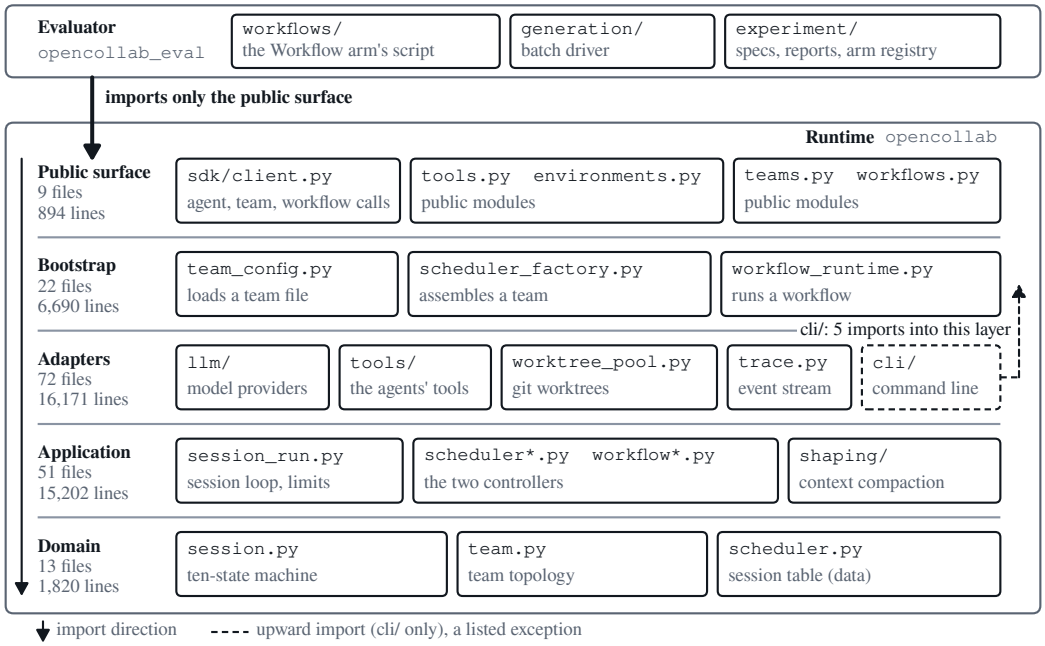

Figure 15: The two code bases. The runtime's layers run top to bottom, each with its file and line counts (Table 12), and a module imports only from its own layer or the ones below; the one exception is five imports from the command-line interface into the bootstrap layer (dashed), which the experiments do not use. The evaluator reaches the runtime only through its public surface.

contracts they violate. Counting import statements directly agrees: every import in the four layers points into its own layer or a lower one, except five statements in three files of the command-line interface, which belongs to the adapters layer, that reach up into the bootstrap layer. The contract file

lists these as known exceptions. The experiments never start the runtime through the command-line interface.

The evaluator. The evaluator's modules import the runtime only through its public surface. An arm is therefore defined by what it passes through that surface, namely role prompts, tool bundles, and a team file or a script, while the session loop and the code that enforces its limits are the runtime's own in every arm. The team files are kept in the runtime's repository, one per configuration. The batch driver (`generation/gen_prediction_batch.py`) runs every arm over the same ordered task list. The `experiment/` package holds the batch specification that fixes a configuration before it is launched, and the per-configuration report that computes adherence and its intervals from the run records.

C.3 Declaring a Collaboration

A team is declared in one file (Figure 2b). The file names the entry role, gives each role a system prompt, the agent profile it starts from, the names of the tools it may call and its token allowance, optionally overrides the role's model, names the context policy every session runs, and lists the directed edges along which one role may message another. Tool names are resolved to implementations from a registry when the team is assembled, so one file yields the same tool set in every run that uses it.

Roles are separated by what they can call, not only by what they are asked to do. A role whose tool set contains no file-editing tool cannot edit through the file tools, whatever its prompt says. A boundary set in the tool set is enforced at call time, so a call outside it is refused rather than left to the model's choice.

The runtime enforces the declaration at the moment of each call. A tool call outside the calling role's tool set, and a message along an edge the file does not declare, are refused, and the refusal is recorded as an event (Appendix C.5).

The edges decide who can reach whom. In the Star team of Figure 4 the entry role and each Coder can message each other, and the two Coders cannot message each other; in the Ring the analyst can message only the coder, the coder only the tester and the tester only the analyst. Every team needs a path back to the entry role: without one, a teammate could not return its result, and a delegation that did happen but could not be returned would look like a model that never delegated.

The roster is fixed before the first model call and cannot change during a run; a request to add an agent is refused and recorded.

C.4 Audit of Ten Agent Artifacts

Section 3.3 names seven conditions for a controlled comparison of agent organizations. Five ask whether a factor can be set explicitly before a run: the model, the tools, the token budget, the context policy and the topology. Two ask whether a run can be checked afterwards against what it was declared to be (*Realized*, *Compared*). Table 1 marks each condition on ten public agent artifacts, with OpenCollab scored by the same rule in the last row. This appendix gives the rule and the reason behind every mark other than ✓.

How the marks were placed. Nine artifacts were read in code at a pinned version; Claude Code is closed-source, and its row rests on vendor documentation. A ✓ records that a researcher sets the factor in a setting the artifact itself reads, such as a configuration file, a declarative specification or a command-line option, and that the artifact applies it to every agent. A – records that the artifact ships a mechanism but takes it only in the researcher's own code, or meets the condition only in part. A × records that no mechanism meets the condition even in part; a setting that only moves the threshold of a built-in behavior or switches it off does not count as one. Each row is read on the mechanism that leaves the organization to the model, such as teams, handoffs, subagents or agents called as tools, as OpenCollab's row is read on its Team controller; a script the researcher fixes before the run is outside the table. Marks follow each artifact's default settings. An artifact that runs a single agent is marked on that agent, and × under *Topology* and *Compared*. A condition is marked ✓ when:

- *Model*: every agent's model can be set, and a declared model is not replaced by another without notice;
- *Tools*: each role can be given its own tool set;
- *Budget*: each model call is admitted against the calling agent's remaining allowance before it is sent, and refused if it does not fit;
- *Context*: a setting chooses the policy that decides what each model call sees;
- *Topology*: the roles and the edges between them are fixed before the run rather than started by the model while it runs;
- *Realized*: each run's record holds the whole declared organization, including roles that never act, with every model call and tool call attributed to its run and agent;
- *Compared*: *Realized* holds, every message between agents names its sender and receiver, and every attempt to leave the declaration is refused or flagged in the record.

The pinned versions and the file-and-line evidence behind every mark are released with the paper.

Why the other rows fall short. On *Model* and *Tools*, AG2 and LangGraph take both only as constructor arguments in the researcher's code, HAL passes a model name on to the agent's own code, and a Codex CLI role file can only switch tools off from a fixed set; HAL leaves the tools to the agent. On *Budget*, no external artifact prices a call against the remaining allowance before sending it. AutoGen, SWE-agent, OpenHands, Claude Code and Inspect AI stop an agent once its limit has been reached or after a call has returned; AG2 only raises alerts, LangGraph limits steps, DeepSeek Harness caps the output of a single call, Codex CLI's token budget sits behind a feature marked as not ready for external use, and HAL bounds a task by wall-clock time. On *Context*, AutoGen and SWE-agent choose the policy in configuration; OpenHands, DeepSeek Harness, AG2, LangGraph and Inspect AI let it be chosen only in part or only in code; Claude Code and Codex CLI expose only a threshold and an off switch, and HAL does not run the agent's loop. On *Topology*, AutoGen and Inspect AI hold the agents and whom each may call in a configuration file; AG2 and LangGraph wire agents in code; in Claude Code, Codex CLI, DeepSeek Harness and OpenHands the roles are declared but the model decides at run time which agents to start, and the edges can be held only by a hook the researcher writes; HAL leaves the organization to the agent's own code. On *Realized*, SWE-agent stores its full configuration with every run; seven rows record only part of the declaration, such as the settings of the agents that were started but not of those that were not, or roles without their models; AutoGen's logs carry no run identifier, and LangGraph's traces hold what each call sent but not the graph. On *Compared*, only Inspect AI refuses or flags departures in its record, and only for agents joined as tools; SWE-agent's record is complete, but it runs a single agent.

Our own row. OpenCollab's row is read on its Team controller. Its team file sets each role's model, tools, token allowance and context policy together with the edges; every call is admitted against the calling role's remaining allowance before it is sent (Appendix C.1); each run's record opens with the declared organization, every role with its model and tools and every edge, so a declared teammate that never acts still appears; and a message along an undeclared edge or a call outside a role's tool set is refused and recorded (Appendix C.5).

C.5 The Run Record

Section 3.4 replaces prose logs with an ordered event stream. This appendix gives the files each configuration writes, the event types, and the recorded termination reasons.

Three files per configuration. The batch manifest holds the assigned condition: arm, task list, role cards with their sha256, per-role allowance, the runtime and evaluator versions, and the driver's environment. The event log holds the realized side, one ordered stream per run. The metrics file holds one row per run, with the image the run was generated in. The run id joins the three files, and every event carries it. Rates are computed from individual runs, never from pre-averaged configurations, and adherence is computed from the manifest and the event log after the run (Appendix C.6).

Table 13: The seven event types. Model calls and agent finishes settle whether a teammate worked; tool executions of `message_agent` show who briefed whom; a session termination, read with the token counts of the model calls, identifies the runs whose decision the allowance cut short.

Event type	What it adds to the common fields	What it supports
Model call	Token counts and latency of the step	Per-role spend; the organizational overhead of Appendix C.5
Tool execution	Tool name, arguments and result	Who briefed whom; which edges were addressed. Refused messages and refused tool calls are written as separate events
Agent finish	Whether the role produced content	Whether a teammate worked, read together with its model calls
Session termination	The role's termination reason (Table 14)	Which runs were cut off by the allowance after delegation began
Refused spawn	The reason for the refusal	That the roster cannot change within a run (Section 3.2); that a single-agent run cannot coordinate
Worktree change	Commit sha	Which role's commits reached which worktree (Section 3.4)
Context shaping	Which compaction stage fired this turn	Whether a call's input was compacted (Section 3.3)

Events. Every event carries a timestamp, a step number that increases within the run, the run id, the event type, and a payload whose fields depend on the type and, for every type the analyses below read, name the role. Table 13 lists the seven types the analyses read and what each one supports; the controllers also write message and budget records. Because the payload names the role, what each role spent, how many steps it took and which tools it called are read directly off the stream, with no prose log to parse.

Cost metrics. Organizational overhead. Multi-agent configurations incur prompt costs even when the assigned organization is not fully realized. Because Table 13 logs token usage by role, OpenCollab measures this prefix tax by comparing, on the same tasks, the cost of Team runs with $A_r = 0$ against the cost of the single-agent baseline. With $C(z)$ and $C(0)$ the potential total costs under target Team configuration z and Single, defined like $Y(z)$ and $Y(0)$ in Appendix A.1,

$$\begin{aligned} \mathrm{Overhead}_0 &:= \mathbb{E}[C(z) - C(0) \mid A(z) = 0], \\ \widehat{\mathrm{Overhead}}_0 &= \frac{1}{n_0} \sum_{i:\, A_i(z)=0} \big(C_i(z) - C_i(0)\big), \end{aligned} \tag{14}$$

where the sum runs over the n_0 tasks whose Team run has $A_r = 0$ and pairs each with the Single run of the same task. This metric is the additional cost of carrying the Team configuration among runs that do not fully realize it. Comparing these Team runs with the unconditional Single mean would instead mix that cost with differences in task composition, since non-adherent runs need not fall on typical tasks.

Workload balance. For fully adherent Team runs ($A_r = 1$), OpenCollab measures how computational effort is divided across the active roles. Letting $p_s(r) = C_s(r)/C(r)$ denote the token proportion consumed by role s, the framework defines workload balance using normalized Shannon entropy across the active roles $\mathcal{S}$:

$$\mathrm{Balance}(r) = -\sum_{s \in \mathcal{S}} \frac{p_s(r) \log p_s(r)}{\log |\mathcal{S}|}. \tag{15}$$

The aggregate balance metric is computed as the expectation of $\mathrm{Balance}(r)$ across all adherent runs where $A_r = 1$. A value near one reflects an egalitarian division of labor, whereas a value near zero indicates that a single role dominated execution despite the presence of teammates.

Table 14: The end states of a session (Figure 12) and the reasons recorded with them. Each reason is a string whose fixed prefix is shown; some continue with a count, such as the tokens used. The wall-clock limit is not among them: the evaluator stops a run that reaches it from outside the runtime and records that in the run's metrics row.

State	Recorded reason	When it is recorded
`DONE`	`completed`; `submitted`	The role gave a final answer, or called the submit tool.
`STOPPED`	`budget exceeded`	At `PRECHECK`, the role had already spent its allowance.
	`budget exhausted before model call`	The next call, priced before it was sent, would not fit in what remained.
	`budget exceeded after model call`	The last call cost more than its estimate and took the role past its allowance.
	`step limit reached`	The session reached its step limit.
	`loop block limit reached`	The role kept repeating a tool call without progress.
	`context overflow`	The prompt exceeded the model's context window even after compaction.
	`output truncated`	The provider cut the response off at its generation limit.
	`required tool was not called after correction`	The role was required to call a tool, was reminded once, and still did not.
	`interrupted by user`; `cancelled`	The session was stopped from outside.
`ERROR`	the exception's type and message	An unhandled fault, including a role whose container or workspace was withdrawn during the run.

C.6 Adherence, Axis by Axis

Section 3.5 defines a run as adherent when every axis that applies to its arm is adherent. This appendix states the six axes (Table 15), how a verdict is reached, and which axes can vary in these experiments.

How a verdict is reached. Each run has an assigned side, written in the configuration's manifest before the run and restated as events in the run's event log, and a realized side, the rest of that log. Each axis compares the two and returns *adherent*, *deviant* or *unverifiable*. $\widehat{\alpha}_{\mathrm{adh}}$ is the share of a configuration's runs that are adherent on every axis that applies. An axis that does not apply to an arm, such as delegation for the single agent, is left out of that arm's conjunction rather than scored as a pass.

Which axes can vary here. An adherent verdict says that a run did not depart from its declaration on that axis, which is not the same as the axis having had anything to measure. Four of the six axes are held by the runtime in these experiments. The runtime offers each agent only the tools of its assigned set and refuses any other call, so no agent can act outside it, which is the one deviation the role-boundary axis records. An allowance refuses the call that would exceed it and records the refusal, so no role can pass its allowance unrecorded, the budget-sharing deviation. A message along an edge the topology does not declare is refused. Every role runs the same context policy. In the Team the conjunction is therefore decided by participation and delegation. A teammate starts working only when a message reaches it (Appendix C.1), so a run in which every declared teammate worked has also delegated, and the conjunction comes down to one test: a Team run is adherent when every declared teammate, every role other than the entry role, spent tokens and produced at least one model output. This is computed for every run from the run record: the roles from the declaration restated in the event log, and each role's tokens and outputs from its saved session. The four held axes still have to be declared: a comparison that gives each model a different tool set, allowance or context window is exactly the case they exist for.

Table 15: The six adherence axes. The assigned value is written before the run, and the realized value is read from the event log.

Axis	Realized value, read from	Deviation
Participation	the teammates that spent tokens and produced at least one model output	a declared teammate did no work
Delegation	a teammate that spent tokens and produced at least one model output	no such teammate, where the assigned regime offers or requires delegation
Role boundary	the tool calls attributed to each agent	a call outside the agent's assigned tool set is executed
Budget sharing	each role's token draw against its own allowance	a role reached its allowance and no refusal was recorded
Information flow	`message_agent` calls in the event log, in every arm	a message travels along an edge the topology does not declare
Context policy	the compaction level applied on each turn	a level other than the assigned one fires

Unverifiable counts against adherence. When the log cannot establish what happened on an axis, the verdict is *unverifiable* and the run is not counted as adherent. This happens, for instance, when a run was cut off after the entry agent had begun to delegate but before every teammate had worked (Appendix B.1 lists the cases). Such a run has $A_r = 0$ (Appendix A.1). Missing evidence can therefore lower a reported rate but never raise it; counting every unverifiable run as adherent instead shows how much the rate depends on this rule.