TEA: STRUCTURALLY REPRESENTING ARBITRARY LLM AGENTS COLLABORATION

Anonymous authors
Paper under double-blind review

ABSTRACT

LLM agent collaboration may take many forms, ranging from predefined or dynamic workflows to open-ended interactions, in which agents create and revise tasks and choose when and with whom to work on the fly. Less structured collaboration lets agents organize their own work and adapt to diverse tasks, environments, and changing teammates, but performance also becomes less reliable and harder to understand. Indeed, even if a workflow is predefined, agents may deviate from it during execution, making tracing each agent's contribution difficult. We introduce Task-Environment-Agents (TEA), a flexible structural representation that makes evolving collaboration explicit without prescribing agent reasoning, behavior, or communication order. TEA couples an Agent Interaction Graph (AIG) to both a Task Activity Graph (TAG) and the environment to form a collaboration state, providing a basis for analysis and optimization. We demonstrate TEA's potential by representing and recovering emergent collaboration structures and analyzing collaboration states. TEA also serves as an active framework through which agents communicate, transform tasks, and act in the environment. We show that TEA enables and improves decentralized agent collaboration on shared tasks.

1 INTRODUCTION

Giving multiple agents a joint task in a shared environment does not automatically mean that they will successfully collaborate. Collaboration emerges when agents *coordinate their contributions* toward accomplishing a task (Guo et al., 2024; Yang et al., 2026b). In particular, LLM (large language model) agents can use pretrained reasoning and tool-use capabilities (Sumers et al., 2024) to create tasks, communicate progress, and divide and carry out work. *Understanding* and *improving* such collaboration require tracking the relationships between agents' interactions, task changes, and effects on the environment without constraining execution.

In practice, LLM agent interactions with each other and the environment are often guided by a *collaboration policy*, e.g., LangGraph's executable graphs (LangChain Inc.) or combinations of code and instructions, as in the Agent Skills open standard (Anthropic, 2025). The policy may contain rules that specify agent responsibilities, such as roles, task assignments, and output schemas; and govern execution patterns, including sequential, parallel, and conditional execution of subtasks, as well as loops. However, collaboration policies may not fully control agent coordination: LLM agents may deviate from their specified policies when actually executing tasks, and the policies themselves may not fully specify agent interactions. *Partially specified policies* provide a structure while leaving some decisions to agents during execution: an orchestrator, for example, may dynamically decompose work and assign tasks to subagents (Xu et al., 2026). In fully *open-ended settings*, agents organize collaboration by themselves on the fly (Yang et al., 2026a;b). While these more flexible policies can lead to superior task performance by taking advantage of LLM agents' ability to adapt to their environment, they make it difficult to track, and thus improve, agent collaboration.

Research challenges. Agents' flexibility in determining their interactions means that collaboration structure may (partially) emerge from agent execution rather than being pre-specified, raising both **representation** and **optimization** challenges. Existing representations often assume turn-based collaboration (Zhang et al., 2025d; Cemri et al., 2026), do not explicitly track agent-created task and its progress (Zhao, 2026; Yang et al., 2026a), or require explicit task validators (Zhang et al., 2026b).

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.

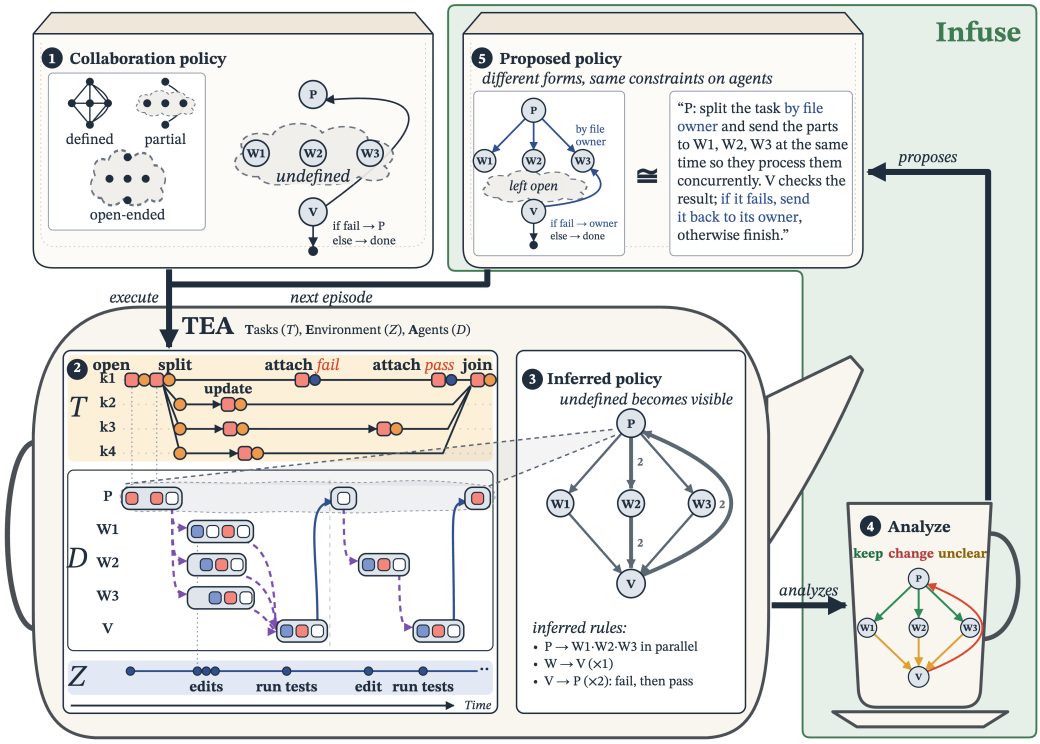


Figure 1: **TEA and Infuse for representing and improving collaboration.** (1) A collaboration policy Γ may fully, partially, or minimally specify coordination, leaving room for agents to make decisions. (2) TEA combines views of **T**ask evolution, **E**nvironmental outcomes, and **A**gent interactions. Solid edges represent behavior prescribed by the policy, while dashed edges represent behavior emerging from agents. (3) Aggregation and trace analysis **recover** and **discover** the topology and candidate execution rules from the actual collaboration process captured in TEA. (4) Infuse queries these records to assess coordination against task progress and environmental outcomes. (5) It proposes an updated policy Γ' to execute.

LLM agents, however, can create and reorganize shared work during execution without fully specified collaboration policies. We need a representation that can ground asynchronous interactions and task evolution in environmental outcomes under combinations of specified and emergent behaviors.

By tying agents' realized collaboration process to task progress and environment dynamics, a structured collaboration representation should enable systematic discovery of better collaboration policies. Existing approaches optimize explicit workflow structures or procedures (Zhuge et al., 2024; Zhang et al., 2025c;a; Liu et al., 2024), but improving collaboration from little or no prescribed coordination requires discovering useful strategies in how agents organize shared work and interactions. A flexible, yet structured collaboration representation provides the search space and reveals the relationships between collaboration actions and task outcomes needed for such a discovery process.

Our contributions. (1) **TEA representation.** We introduce *Task-Environment-Agents (TEA)*, which connects an event-based model of asynchronous agent interactions to a task graph that agents create and modify, grounding both in the shared environment. TEA preserves task identities and versions, attributes task changes to agent activations, and lets agents attach evidence to task versions as signals of progress. (2) **Representation capabilities.** We formally define collaboration policies and show that TEA can represent a large class of agent interactions. We then show how TEA supports policy recovery and discovery through topology reconstruction and inference of candidate execution rules, alongside structural queries across its layers. (3) **Representation experiments.** We demonstrate how TEA represents diverse interaction and task structures and correctly recovers their topology and policy properties. (4) **Improving open-ended collaboration.** TEA also serves as an interface for agent collaboration. We introduce *Infuse*, which learns from open-ended collaboration

by recovering a collaboration policy from TEA traces, assessing the traces through queries against task progress and environmental outcomes, and updating collaboration policies.

After reviewing related work (Section 2), Section 3 defines TEA for representing collaboration. Section 4 examines policy recovery and structural analysis. Section 5 presents *Infuse* for iterative policy improvement. Section 6 reports the experimental results, and Section 7 concludes the paper.

2 Related Work

LLM multi-agent collaboration. LangGraph and AutoGen support orchestration and communication (LangChain Inc.; Wu et al., 2024), while CAMEL and MetaGPT organize agent roles and workflows (Li et al., 2023; Hong et al., 2024). Collaboration policies can take the form of executable graphs, as in LangGraph, or code and instructions, as in Agent Skills (Anthropic, 2025). They may allow dynamic task decomposition (Xu et al., 2026) or leave coordination to agents (Yang et al., 2026a;b). Our TEA representation captures both specified and emergent coordination.

Collaboration optimization. GPTSwarm and AFlow optimize graphs or workflow code using run feedback (Zhuge et al., 2024; Zhang et al., 2025c). MaAS optimizes prompts and topologies (Zhang et al., 2025a), DyLAN adapts teams (Liu et al., 2024), and other methods prune redundant communication (Zhang et al., 2025b). These methods refine explicit structures or defined coordination procedures. Infuse learns coordination guidance under partially specified policies, using TEA traces to assess candidate strategies against task progress and environmental outcomes.

Collaboration representation and analysis. Existing LLM-agent traces assume turn-based activity and produce turn-based trajectories (Cemri et al., 2026; Zhang et al., 2025d), which do not account for asynchronous, overlapping agent activity. Langfuse nests calls (Langfuse), but nesting alone does not establish information flow. GRADE separates execution order from information dependencies (Zhao, 2026), and DIG captures asynchronous activations and exchanges (Yang et al., 2026a); neither explicitly tracks agent-created work and its progress. iCORE links interactions to evolving obligations and evidence but requires predefined validation rules (Zhang et al., 2026b). TEA connects task identities and versions to agent activations, evidence, and environmental outcomes, supporting analysis and recovery of existing collaboration policies and discovery of better ones.

3 Task-Environment-Agents (TEA)

TEA connects asynchronous agent interactions to the tasks agents create and revise and to outcomes in their shared acting environment. Let DT denote the TEA representation of a collaboration process:

$$\mathsf{DT} = (\mathcal{D}, \mathcal{T}, \mathcal{Z}, \Phi, \mathcal{U}). \tag{1}$$

Here $\mathcal{D}$ is the Agent Interaction Graph, $\mathcal{T}$ is the Task Activity Graph, $\mathcal{Z}$ is the shared environment, Φ is the evidence attachment set, and $\mathcal{U}$ is the tool space. We introduce these components below; full definitions and proofs appear in Appendices A and B.

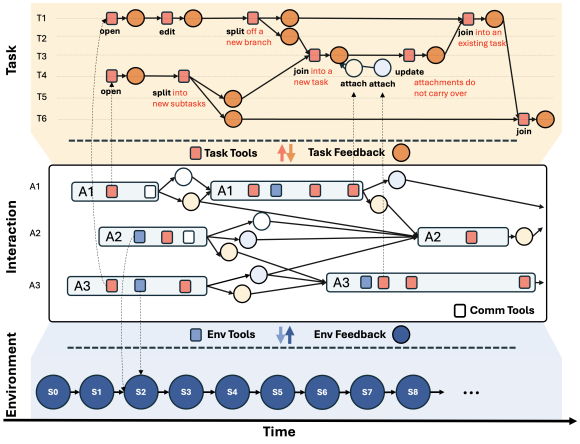


Figure 2: **TEA**. Task layer is the TAG $\mathcal{T}$, Interaction layer is the AIG $\mathcal{D}$, and Environment layer is $\mathcal{Z}$.

Agent Interaction Graph (AIG). The AIG $\mathcal{D} = (\mathcal{E}, \mathcal{H}, \mathcal{L}_D)$ is a directed bipartite graph, where $\mathcal{E}$ is the event, $\mathcal{H}$ the activation, and $\mathcal{L}_D$ the edge set. An activation $h \in \mathcal{H}$ is one reasoning cycle of an agent i_h. It records the reasoning interval, received events, and ordered tool calls with their results (Figure 3). Events, which are produced by activations, can contain any payload, including tool results and messages. In the Interaction layer of Figure 2, wide rectangles are activations, inner squares are tool calls, and circles are events. AIG gives the following representation property.

Proposition 1 (DAG sufficiency of AIG). *AIG supports* $\{e_i\} \to h \to \{e'_j\}$ *and* $e \to \{h'_k\}$, *with each output event having exactly one parent activation. With inputs available at activation start,*

it forms a temporal DAG (directed acyclic graph) that can represent any finite DAG of activation dependencies. Timestamps consistent with these dependencies are recorded as annotations.

Remark 1 (Representing agent interaction)**.** *Events produced by earlier or ongoing activations can wake agents and trigger new activations. Their execution intervals may overlap, while causal order naturally emerges from event production and receipt. These interactions form a temporal DAG over activation occurrences and events, which is precisely the structure represented by the AIG.*

TAG: Task Activity Graph. The TAG $\mathcal{T} = (\mathcal{Q}, \mathcal{A}, \mathcal{L}_T)$ is a directed bipartite graph, where $\mathcal{Q}$ is the task-version set, $\mathcal{A}$ is the task-action set, and $\mathcal{L}_T$ is the edge set. A task version is $q = (k, \delta, \sigma) \in \mathcal{Q}$, where k is its persistent task identity, δ specifies its goal and evaluation rule, and σ is its state. Versions are immutable; a revision to a task creates a new version. Each action $a \in \mathcal{A}$ records a task tool call issued by an agent activation. Its incoming edges identify existing input versions, and its outgoing edges identify newly created output versions. In the Task layer of Figure 2, horizontal rows correspond to task identities, orange circles represent versions, and squares represent actions. Table 1 summarizes the tool calls that capture how agents act on tasks. CLOSE propagates closure backward through recorded ancestry. At a SPLIT, closure reaches the parent only when all tasks produced by that split are closed.

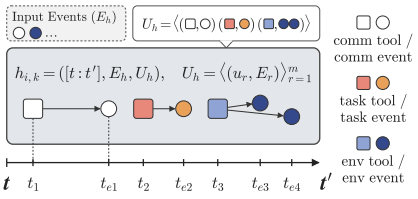


Figure 3: In AIG, agent i's kth activation $h_{i,k}$ records inputs, duration, and ordered tool calls with returned events.

Task actions connect existing versions to newly created versions, preserving a history as work branches and merges, giving the following representation property.

Table 1: Task tools in $\mathcal{U}_T$. An edge $q \to q'$ means that an action uses q to create q'.

Task tool	TAG effect
OPEN(δ, σ)	$\varnothing \to q$
EDIT$_\delta(q, \delta')$	$q \to q'$
UPDATE$_\sigma(q, \sigma')$	$q \to q'$
SPLIT$(q, \{(k_j, \delta_j, \sigma_j)\}_{j=1}^m)$	$q \to \{q_1, \ldots, q_m\}$
JOIN$(\{q_j\}_{j=1}^m, k', \delta', \sigma')$	$\{q_1, \ldots, q_m\} \to q'$
CLOSE(k)	propagate closure

Proposition 2 (DAG sufficiency of TAG tools)**.** *Table 1's tools can represent any finite DAG of task versions. The TAG construction keeps the original dependencies unchanged, adding at most one auxiliary leaf for each version with one predecessor. Timestamps are recorded as annotations.*

Remark 2 (Representing task evolution)**.** *Task dependencies are commonly represented as DAGs (Deelman et al., 2015), and version-control systems such as Git record development history as a DAG of commits (Loeliger & McCullough, 2012). As agents generate and process tasks, their actions link existing task versions to new ones, naturally forming a temporal DAG that preserves the history of task processing. This is precisely the structure represented by TAG.*

TEA: TAG, Environment, AIG. TEA couples the AIG $\mathcal{D}$, TAG $\mathcal{T}$, and environment $\mathcal{Z}$ through tool calls and evidence attachments. In addition to task and environment tools, it provides communication and evidence tools $\mathcal{U}_\phi = \{\text{send}_h, \text{attach}_h\}$. The tool call $\text{send}_h(e, J)$ sends an event e to agents in $J \subseteq V$. The sending agent can construct a new message, or forward an existing event. The tool call $\text{attach}_h(\phi)$ adds this attachment to Φ. Evidence remains attached to its target task version and does not carry over to later task revisions.

Table 2: TEA transitions.

Tool call	AIG	TAG	Env.	Evidence
$u \in \mathcal{U}_T$	$\mathcal{D}'$	$\mathcal{T}'$	$\mathcal{Z}$	Φ
$u \in \mathcal{U}_Z$	$\mathcal{D}'$	$\mathcal{T}$	$\mathcal{Z}'$	Φ
$\text{attach}_h(\phi)$	$\mathcal{D}'$	$\mathcal{T}$	$\mathcal{Z}$	Φ'
$\text{send}_h(e, J)$	$\mathcal{D}'$	$\mathcal{T}$	$\mathcal{Z}$	Φ

Every tool call updates the AIG by recording the tool call in its issuing activation h (Figure 3), along with the returned event set E_u. Task and environment calls additionally act on TAG and the environment, respectively. Table 2 summarizes these effects: primes indicate an update.

Remark 3 (Evidence and task progress)**.** *For a task version q, $\Phi_q = \{(h, x, q) \in \Phi\}$ is the set of evidence attached to it. Interpretations from different agents may agree, partially agree, or disagree. The evidence set, its providers, and its content together provide signals of task progress. Different interpreters at different times may assign different weights to evidence from distinct sources.*

4 EXPLORING THE POTENTIAL OF TEA

We explore TEA's potential to (i) recover collaboration structure and policy properties from execution records, thus demonstrating its ability to represent collaboration structures; and (ii) analyze relationships among agent interactions, task evolution, environment activity, and evidence.

We first formally define a collaboration policy for LLM agents. Each LLM agent follows its own policy to decide what to do based on its available information and private state. The agents' decisions and message exchanges (captured by AIG's activations and events) produce their collective behavior. For a given team, task, and setup, this behavior follows a joint distribution over **when** agents activate, **what** they receive, **which** tools they call, and **what** they send to **whom** and **when**. A collaboration policy Γ shapes this distribution through the form and degree of its coordination specifications and the freedom it leaves agents to organize collaboration on the fly.

Definition 1 (Collaboration policy for LLM agents)**.** *For a fixed agent team, task, and execution setup, a collaboration policy* Γ *is an interaction topology along with a set of coordination rules that together guide or constrain task execution, shaping the joint distribution of agent activations, tool calls, and event exchanges over time.*

4.1 RECOVERING/DISCOVERING COLLABORATION POLICIES

We recover the observed topology of agent interactions and infer candidate execution and coordination rules from TEA records.

Topology recovery through activation aggregation. Let S be the set of activations in AIG that start and finish within the time interval $[t, t']$: $S = \{h \in \mathcal{H} : t \le s_h \le f_h \le t'\}$, where s_h and f_h are activation h's start and finish times. Let V be the set of agents and X_{ij} the list of messages sent to agent j by agent i's activations in S. We recover the agent topology by merging activations by agent, giving the weighted graph $G_{[t,t']} = (V, E, w)$, with edges $E = \{(i, j) : |X_{ij}| > 0\}$ and message frequency counts $w_{ij} = |X_{ij}|$ as weights. The full agent topology is recovered when the selected records cover every possible edge created under a given collaboration policy Γ.

Inferring execution rules. Starting from $G_{[t,t']}$, we examine activation times, send and receipt records, and recorded decisions in the corresponding TEA trace. We group consecutive activations whose starts fall within τ of each other and whose intervals share a common overlap. For distinct completed activations on a common clock, *execution order* and *overlap* are formally defined by

$$h \prec h' \iff f_h \le s_{h'}, \quad h \parallel_\tau h' \iff |s_h - s_{h'}| \le \tau. \quad (2)$$

Table 3: Example trace.

Activation	Interval	Receives from
$h_{a,1}$	$[0, 1]$	Task input
$h_{b,1}$	$[1, 4]$	$h_{a,1}$
$h_{c,1}$	$[1.5, 7]$	$h_{a,1}$
$h_{a,2}$	$[4, 5]$	$h_{b,1}$
$h_{b,2}$	$[5, 6]$	$h_{a,2}$
$h_{a,3}$	$[7, 8]$	$h_{b,2}, h_{c,1}$
$h_{c,2}$	$[8, 10]$	$h_{a,3}$
$h_{a,4}$	$[10, 11]$	$h_{c,2}$

In Table 3's example, $h_{i,k}[s, f]$ denotes agent i's kth activation over $[s, f]$. Aggregation gives $a \leftrightarrow b$ and $a \leftrightarrow c$, with two messages per direction. With $\tau = 0.6$, $h_{b,1}$ and $h_{c,1}$ are grouped together, giving the following order of activation starts: $\{a\} \to \{b, c\} \to \{a\} \to \{b\} \to \{a\} \to \{c\} \to \{a\}$. We can then deduce the following execution rules:

(1) Sequential execution. Apply $\prec$ along the activation paths. In our example, both $h_{a,1} \prec h_{c,1} \prec h_{a,3}$ and $h_{a,3} \prec h_{c,2} \prec h_{a,4}$ support the local agent order $a \to c \to a$.

(2) Parallel execution. Applying $\parallel_\tau$ shows that $h_{c,1}$ overlaps $h_{b,1}$, $h_{a,2}$, and $h_{b,2}$. Agent a can therefore receive b's result and send more work to b while c remains active.

(3) Repetition. Follow message paths that return to later activations of the same agent. The path $h_{a,1} \to h_{b,1} \to h_{a,2} \to h_{b,2} \to h_{a,3}$ contains two occurrences of $a \to b \to a$, identifying repeated interaction in the trace.

(4) Conditional execution. The graph shows whom an agent sends to, but the conditions behind each message must be inferred from its input messages and subsequent sending behavior.

(5) Joins. A join brings execution branches together. We infer its rule from observing which senders provide inputs to an agent's activations and how consistently these sets recur. For example, $h_{a,3}$ receives from both b and c. Repeated activations receiving both under the same conditions support an all-input rule; receiving either alone supports an any-input rule.

4.2 Analyzing Relationships in TEA

Agents are the common actors across AIG, TAG, and the environment. Their activities induce directed graph edges ($\rightarrow$) and implicit links ($\mapsto$) in TEA; for example, a TAG tool call is recorded in AIG and creates a corresponding action node in TAG. Table 4 summarizes these relations.

Paths and queries. We analyze these relationships through *queries*, which traverse TEA's edges and links in either direction, within or across layers. Each path between two entities gives an interpretation of their relationship based on its link types and traversal directions. For example, tracing task versions can reveal who worked on a task, what they changed and in what order, and the linked evidence and environmental feedback. Tracing further back to agent activations shows whom they interacted with and what information they received, helping explain their actions.

Table 4: Relations and tools in TEA.

AIG interaction	Actor	$(h, u) \mapsto h \mapsto i_h$
	Event flow	$e \rightarrow h$; $h \rightarrow e'$
TAG task tools ($\mathcal{U}_T$)	Identity	$q \mapsto k$
	Open / Close	$a \rightarrow q$; $a \mapsto k$
	Edit / Update	$q \rightarrow a \rightarrow q'$
	Split	$q \rightarrow a \rightarrow \{q'_j\}$
	Join	$\{q_j\} \rightarrow a \rightarrow q'$
TEA tools and links	Task action	$(h, u) \mapsto a, u \in \mathcal{U}_T$
	Environment	$(h, u) \mapsto u, u \in \mathcal{U}_Z$
	send_h	$(i_h, e, j), j \in J$
	attach_h	$(h, u) \mapsto \phi$
	Feedback	$(h, u) \mapsto E_u \subseteq E$
	Evidence	$\phi \mapsto h$; $\phi \mapsto q$; $\phi \mapsto x$

5 Infuse

We instantiate TEA as a framework for agent collaboration and use *Infuse* to analyze the resulting execution trace and propose collaboration policy.

Collaboration framework. A team of agents V collaborates asynchronously through TEA. For each agent $i \in V$, the functions $f_{\text{activate}}(i)$ and $f_{\text{deactivate}}(i)$ start and end its kth activation $h_{i,k}$, respectively. During an activation, the agent acts on $o_i = (o_{\mathcal{Z},i}, o_{\mathcal{T},i}, M_i, \eta_i)$, where $o_{\mathcal{Z},i}$ is its environment view, $o_{\mathcal{T},i}$ is its TAG view, M_i contains received messages, and η_i is its private memory. Its tool calls support communication, task transformations, evidence attachment, and environmental actions, all recorded through TEA.

TEA provides three mechanisms to control agent activation. (1) After each round of reasoning and tool use, the agent decides whether to continue its current activation or deactivate. (2) The `sleep` operation ends the current activation and schedules reactivation after a specified delay. (3) When sending a message, the agent specifies whether it should interrupt the recipient. Delivery activates an inactive recipient. If the recipient is active and interruption is requested, its current activation ends and a new one begins with the incoming message.

Infuse. We consider improving the task performance of an LLM agent team in an open-ended setting: agents collectively attempt a task starting from an empty collaboration policy Γ (Definition 1). *Infuse* refines how the agents collaborate by analyzing the resulting execution records and proposing an updated policy (Algorithm 1). TEA grounds the AIG and TAG in environmental outcomes by linking agent interactions and task updates to recorded environmental activity and feedback. Infuse queries this representation to check the recovered policy against the execution trace and assess its components, guiding what to keep, add, revise, or leave open. In our implementation, GPT-6-Sol consolidates this guidance into global lessons for the team and private lessons for individual agents, forming the proposed policy Γ'. However, Infuse itself is agnostic to the algorithm for producing Γ'.

Algorithm 1 *Infuse*

Require: Agents V, policy Γ
Execute and record collaboration
1: $\mathsf{DT} \leftarrow f_{\text{execute}}(V, \Gamma)$
Recovery/discovery: Sec. 4.1
2: $\widehat{\Gamma} \leftarrow f_{\text{recover}}(\mathsf{DT})$
Trace queries: Sec. 4.2
3: $\Gamma' \leftarrow f_{\text{analyze}}(\Gamma, \widehat{\Gamma}, \mathsf{DT})$

6 Experiments

We conduct two types of experiments. **Representation experiments** test whether TEA preserves agent interactions and task histories and supports coordination recovery. **Empirical experiments**

test whether TAG improves open-ended collaboration and whether Infuse can discover and improve a collaboration policy.

6.1 Representation experiments

We implement different collaboration policies in LangGraph by designing workflows that specify interaction graphs, coordination rules, and how agents process tasks. We then execute these workflows and visualize the resulting TEA traces.

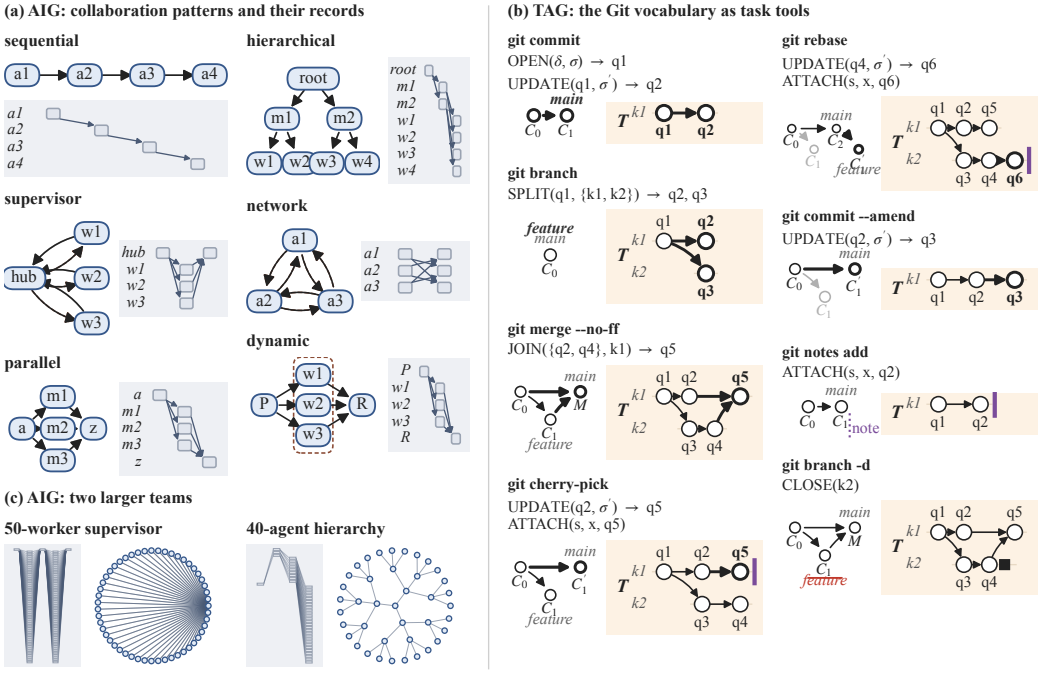


Figure 4: (a) Interaction graphs and AIG traces for six collaboration patterns. (b) TAG is a general-purpose task representation that expresses Git's collaborative coding semantics through individual or composed actions. (c) AIG represents teams of any size, illustrated by a 50-worker supervisor and a 40-agent hierarchy.

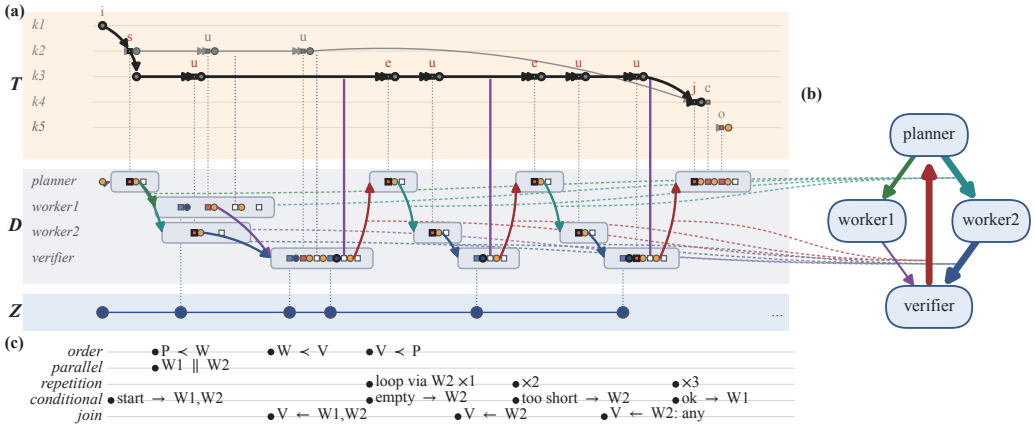


Figure 5: (a) Task evolution ($\mathcal{T}$), agent activity ($\mathcal{D}$), and environment transitions ($\mathcal{Z}$). (b) Interaction graph recovered from the record. (c) Execution rules inferred as the record grows (P: planner; W: workers; V: verifier).

Interactions and task history. Figure 4 shows the distinct AIG traces produced by different LangGraph patterns during execution. The Git examples show how individual or composed TAG actions

represent revisions, branches, and merges. Figure 5 shows an execution trace generated by a hidden collaboration policy. Following Section 4.1, we aggregate activations in the AIG trace by agent to recover the interaction topology, with edges weighted by interaction frequency.

Queries. In Figure 5, the inferred rules show repetition and execution order. Checking TAG states helps explain conditional routing. We can also ask "Who worked on task k_2, and why?" The record shows that the planner creates k_2 by splitting k_1 and sends messages to both workers. In the first round, worker1 is the only worker that acts in the environment. It updates k_2 and then sends a message to the verifier.

Changes over time. A task may have multiple milestones that benefit from different collaboration policies. Figure 6 shows an example in which the policy changes across three phases. Aggregating interactions over reasonable chunks helps recover the policy in each phase. However, chunks that cross phases could greatly affect the recovered interaction topology by mixing interactions from different policies. This opens up new opportunities in detecting policy changes and selecting aggregation windows.

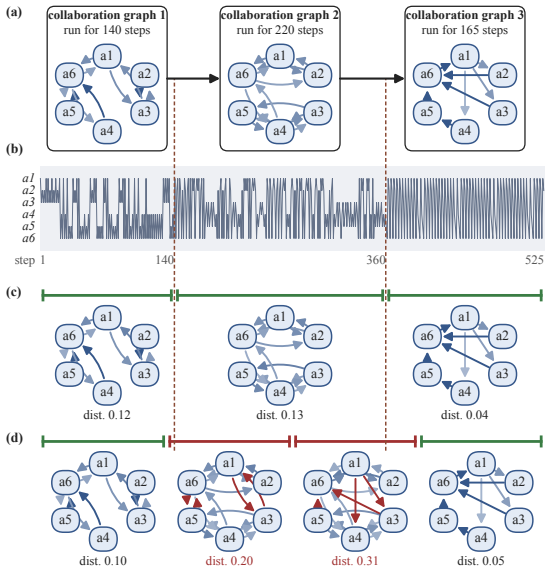


Figure 6: (a) Collaboration policies for three milestones and (b) their execution trace. (c) Perfect chunking and (d) equal-length chunking, with their recovered policies.

6.2 Empirical Experiments

We empirically evaluate task performance in two domains, code refactoring and Silo-Bench (Zhang et al., 2026a), using TEA as an open-ended collaboration framework and Infuse to improve the collaboration policy. Both domains use five `gpt-5.6-terra` agents without predefined roles. A separate `gpt-6-sol` agent derives Infuse guidance from execution records and outcomes.

(1) Code refactoring. We define a task where agents modularize a correct but redundant codebase, with performance measured by LoC reduction while passing fixed tests. We evaluate six repositories at three difficulty levels based on repository size. Agents share all files but may modify only source code, using `list_files`, `read_file`, `edit_file`, `measure`, and `run_tests`. Each run has a budget of 100 total agent steps. We report final test pass rates and LoC reduction. **(2) Silo-Bench.** We evaluate 30 distributed reasoning tasks across three difficulty levels. Each agent accesses local information through `read_data` and uses `submit` to answer a query requiring information from across the team. We report the task success rate (all agents correct) and mean S and P, where S is the fraction of exactly correct answers and P averages scores allowing partial credit.

6.2.1 Understanding the Performance of TEA as a Collaboration Framework

To understand TEA's contribution to collaboration, we ablate access to TAG and direct communication in four conditions: TAG + comm, TAG only, communication only, and neither.

Code refactoring. Without TAG, communication only and neither achieve similar reductions among passing runs: 19.5% and 20.8% (Table 5). TAG increases reduction to 35.2% without communication and 43.0% with it. TAG + comm achieves the largest reduction. Figure 7 shows the reduction gap widening during execution. Curves show the mean across five runs per condition, with shading indicating $\pm$ standard deviation.

Silo-Bench. With neither TAG nor direct communication, the success rate is 0.03. Communication alone reaches 0.60, compared with 0.77 for both TAG conditions (Table 5). TAG provides a shared task record for implicit communication and synchronization, which may help agents coordinate their answers. TAG + comm further improves partial correctness (0.85 versus 0.83). *TEA's shared task*

Table 5: Code refactoring on `shop` (left): five runs per condition. Silo-Bench (right): 30 tasks. Pass counts runs passing all tests; R_{all} and R_{pass} are mean LoC reductions (%) over all and passing runs. Best values are bolded.

	Code (`shop`)			Silo-Bench		
Setting	Pass ↑	R_{all} ↑	R_{pass} ↑	Success ↑	S ↑	P ↑
TAG + comm	**4/5**	**46.9**	**43.0**	**0.77**	**0.81**	**0.85**
TAG only	3/5	35.3	35.2	**0.77**	**0.81**	0.83
Comm only	3/5	20.2	19.5	0.60	0.73	0.78
Neither	**4/5**	21.3	20.8	0.03	0.15	0.20

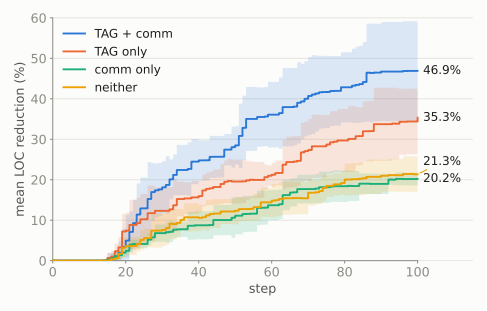


Figure 7: LoC reduction over agent steps on `shop`. Shading shows standard deviation of 5 runs.

graph outperforms direct communication alone on both benchmarks, while combining the two yields the strongest overall results.

6.2.2 Infuse to Discover and Improve Collaboration Policy

Infuse uses a separate `gpt-6-sol` agent to generate global and agent-specific lessons and refactoring file assignments from prior lessons, logs, and outcomes. We compare TEA performance with and without Infuse.

Code refactoring. After one Infuse update, the average pass rate rises from 0.63 to 0.70 (Table 6). Mean LoC reduction among passing runs increases in all six repositories, with the repository average rising from 25.5% to 30.4%. Pass rates rise on `ledger`, `freight`, and `stay`, remain unchanged on `shop` and `plan`, and only fall on `payroll`.

Table 6: Code-refactoring results before (TEA) and after one Infuse update (Inf.), with five runs per repository and condition. Average is the unweighted repository mean; ± denotes sample SD across passing runs, or repositories for Average. Better paired values are bolded; ties are unbolded.

Level	Easy				Medium				Hard					
Codebase	`shop`		`plan`		`ledger`		`freight`		`stay`		`payroll`		Average	
	TEA	Inf.	TEA	Inf.	TEA	Inf.	TEA	Inf.	TEA	Inf.	TEA	Inf.	TEA	Inf.
Pass Rate ↑	0.80	0.80	0.80	0.80	0.60	**0.80**	0.60	**0.80**	0.40	**0.60**	**0.60**	0.40	0.63	**0.70**
LoC Redu. (%) ↑	43.0 ±10.2	**56.6** ±8.9	28.3 ±5.9	**31.6** ±3.7	22.1 ±3.4	**25.1** ±1.7	20.6 ±4.0	**22.1** ±3.5	20.2 ±2.6	**24.8** ±4.4	18.8 ±2.5	**22.2** ±9.0	25.5 ±9.2	**30.4** ±13.3

Silo-Bench. Infuse increases success rates in three of four settings (Table 7). With TAG + comm, success rises from 0.77 to 0.83 and exact correctness from 0.81 to 0.86. Communication-only success rises from 0.60 to 0.67. With neither TAG nor communication, success rises from 0.03 to 0.17 but remains below the other settings. TAG-only success slightly falls from 0.77 to 0.73. *Infuse enables TEA to refine its collaboration policy from experience and improve overall task performance.*

Table 7: Silo-Bench before (TEA) and after one Infuse update (Inf.) on 30 tasks. Bold marks the best value in each row.

	TAG + comm		TAG only		Comm only		Neither	
	TEA	Inf.	TEA	Inf.	TEA	Inf.	TEA	Inf.
Success ↑	0.77	**0.83**	0.77	0.73	0.60	0.67	0.03	0.17
S ↑	0.81	**0.86**	0.81	0.75	0.73	0.79	0.15	0.19
P ↑	0.85	**0.88**	0.83	0.78	0.78	0.82	0.20	0.24

7 Conclusion

We introduce TEA to represent and analyze collaboration by connecting agent interactions, task versions, and environmental outcomes. We define a collaboration policy as an interaction topology and a set of coordination rules that together guide how collaboration unfolds. When the policy is unknown, TEA supports discovery of the observed topology and inference of candidate rules. TEA also serves as a framework for open-ended collaboration. TEA also serves as a general representation and framework for open-ended collaboration, grounding agent-generated tasks and interactions in environmental outcomes.

AI USE STATEMENT

We use large language models to polish the language, help implement TEA as an executable collaboration framework, and create code-refactoring tasks across six repositories of varying sizes. The authors remain involved throughout the process and have reviewed and verified all AI-assisted outputs. We take responsibility for the final content of this work.

REFERENCES

Anthropic. Skills for organizations, partners, the ecosystem. https://claude.com/blog/organization-skills-and-directory, 2025.

Mert Cemri, Melissa Z Pan, Shuyi Yang, Lakshya A Agrawal, Bhavya Chopra, Rishabh Tiwari, Kurt Keutzer, Aditya Parameswaran, Dan Klein, Kannan Ramchandran, et al. Why do multi-agent llm systems fail? *Advances in Neural Information Processing Systems*, 38, 2026.

Ewa Deelman, Karan Vahi, Gideon Juve, Mats Rynge, Scott Callaghan, Philip J Maechling, Rajiv Mayani, Weiwei Chen, Rafael Ferreira Da Silva, Miron Livny, et al. Pegasus, a workflow management system for science automation. *Future Generation Computer Systems*, 46:17–35, 2015.

Taicheng Guo, Xiuying Chen, Yaqi Wang, Ruidi Chang, Shichao Pei, Nitesh V. Chawla, Olaf Wiest, and Xiangliang Zhang. Large language model based multi-agents: A survey of progress and challenges, 2024. URL https://arxiv.org/abs/2402.01680.

Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, Chenyu Ran, Lingfeng Xiao, Chenglin Wu, and Jürgen Schmidhuber. MetaGPT: Meta programming for a multi-agent collaborative framework. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=VtmBAGCN7o.

LangChain Inc. LangGraph. GitHub repository. URL https://github.com/langchain-ai/langgraph.

Langfuse. Langfuse: Open source agent evals & observability. https://langfuse.com/.

Guohao Li, Hasan Hammoud, Hani Itani, Dmitrii Khizbullin, and Bernard Ghanem. Camel: Communicative agents for" mind" exploration of large language model society. *Advances in neural information processing systems*, 36:51991–52008, 2023.

Zijun Liu, Yanzhe Zhang, Peng Li, Yang Liu, and Diyi Yang. A dynamic LLM-powered agent network for task-oriented agent collaboration. In *First Conference on Language Modeling*, 2024. URL https://openreview.net/forum?id=XII0Wp1XA9.

Jon Loeliger and Matthew McCullough. *Version Control with Git: Powerful tools and techniques for collaborative software development*. " O'Reilly Media, Inc.", 2012.

Theodore Sumers, Shunyu Yao, Karthik R Narasimhan, and Thomas L. Griffiths. Cognitive architectures for language agents. *Transactions on Machine Learning Research*, 2024. ISSN 2835-8856. URL https://openreview.net/forum?id=1i6ZCvflQJ. Survey Certification, Featured Certification.

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*, 2024. URL https://openreview.net/forum?id=BAakY1hNKS.

Iris Xu, Guangtao Zeng, Zexue He, Charles Jin, Aldo Pareja, Dan Gutfreund, Chuang Gan, and Zhang-Wei Hong. BOAD: Discovering hierarchical software engineering agents via bandit optimization. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=O6stE173BD.

Hanqing Yang, Hyungwoo Lee, Yuhang Yao, Zhiwei Liu, Kay Liu, Jingdi Chen, and Carlee Joe-Wong. Dig to heal: Scaling general-purpose agent collaboration via explainable dynamic decision paths. *arXiv preprint arXiv:2603.00309*, 2026a.

Hanqing Yang, Narjes Nourzad, Shiyu Chen, Marie Siew, Jingdi Chen, and Carlee Joe-Wong. Coop ?2: Defining, observing, and repairing cooperation in llm multi-agent systems. *arXiv preprint arXiv:2603.00349*, 2026b.

Guibin Zhang, Luyang Niu, Junfeng Fang, Kun Wang, Lei Bai, and Xiang Wang. Multi-agent architecture search via agentic supernet. *arXiv preprint arXiv:2502.04180*, 2025a.

Guibin Zhang, Yanwei Yue, Zhixun Li, Sukwon Yun, Guancheng Wan, Kun Wang, Dawei Cheng, Jeffrey Xu Yu, and Tianlong Chen. Cut the crap: An economical communication pipeline for LLM-based multi-agent systems. In *The Thirteenth International Conference on Learning Representations*, 2025b. URL https://openreview.net/forum?id=LkzuPorQ5L.

Jiayi Zhang, Jinyu Xiang, Zhaoyang Yu, Fengwei Teng, Xionghui Chen, Jiaqi Chen, Mingchen Zhuge, Xin Cheng, Sirui Hong, Jinlin Wang, et al. Aflow: Automating agentic workflow generation. In *International Conference on Learning Representations*, volume 2025, pp. 34040–34077, 2025c.

Shaokun Zhang, Ming Yin, Jieyu Zhang, Jiale Liu, Zhiguang Han, Jingyang Zhang, Beibin Li, Chi Wang, Huazheng Wang, Yiran Chen, and Qingyun Wu. Which agent causes task failures and when? On automated failure attribution of LLM multi-agent systems. In Aarti Singh, Maryam Fazel, Daniel Hsu, Simon Lacoste-Julien, Felix Berkenkamp, Tegan Maharaj, Kiri Wagstaff, and Jerry Zhu (eds.), *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 76583–76599. PMLR, 13–19 Jul 2025d. URL https://proceedings.mlr.press/v267/zhang25cq.html.

Yuzhe Zhang, Feiran Liu, Yi Shan, Xinyi Huang, Xin Yang, Yueqi Zhu, Xuxin Cheng, Cao Liu, Ke Zeng, Terry Jingchen Zhang, and Wenyuan Jiang. SILO-BENCH: A scalable environment for evaluating distributed coordination in multi-agent LLM systems. In Maria Liakata, Viviane P. Moreira, Jiajun Zhang, and David Jurgens (eds.), *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 29379–29398, San Diego, California, United States, July 2026a. Association for Computational Linguistics. ISBN 979-8-89176-390-6. doi: 10.18653/v1/2026.acl-long.1354. URL https://aclanthology.org/2026.acl-long.1354/.

Zuyuan Zhang, Hanqing Yang, Carlee Joe-Wong, and Tian Lan. Auditing emergent llm-agent collaboration through cooperation-obligation coupling. *arXiv preprint arXiv:2607.27429*, 2026b.

Yue Zhao. Grade: Graph representation of llm agent dependency and execution. *arXiv preprint arXiv:2606.22741*, 2026.

Mingchen Zhuge, Wenyi Wang, Louis Kirsch, Francesco Faccio, Dmitrii Khizbullin, and Jürgen Schmidhuber. GPTSwarm: Language agents as optimizable graphs. In *Forty-first International Conference on Machine Learning*, 2024. URL https://openreview.net/forum?id=uTC9AFXIhg.

A Full Definition of TEA

Agents activate asynchronously to receive and send information events to and from each other, perform private computations, and act in the environment. TEA unifies their interactions with other agents, tasks, and the environment. Let DT denote the TEA representation of a collaboration process:

$$\begin{gathered}\mathsf{DT} = (\mathcal{D}, \mathcal{T}, \mathcal{Z}, \Phi, \mathcal{U}), \quad \text{where} \\ \mathcal{D} = (\mathcal{E}, \mathcal{H}, \mathcal{L}_D), \qquad \mathcal{T} = (\mathcal{Q}, \mathcal{A}, \mathcal{L}_T), \qquad \mathcal{U} = \mathcal{U}_T \cup \mathcal{U}_Z \cup \mathcal{U}_\phi.\end{gathered} \tag{3}$$

Here $\mathcal{D}$, $\mathcal{T}$, and $\mathcal{Z}$ are the Agent Interaction Graph, Task Activity Graph, and shared environment, respectively. The set Φ contains evidence attachments linking agent activations and events to task versions, and $\mathcal{U}$ is the tool space. These components are defined below.

A.1 AIG: Agents Interaction Graph.

The AIG $\mathcal{D}$ is a directed bipartite graph of agent activation and information event nodes. The k-th activation of agent $i \in \mathcal{I}$, denoted $h_{i,k} \in \mathcal{H}$, records

$$h_{i,k} = ([t : t'], E_h, U_h), \quad U_h = \langle (u_r, E_r) \rangle_{r=1}^{m}, \quad u_r \in \mathcal{U}. \tag{4}$$

Here $[t : t']$ is the activation's execution time interval, $E_h \subseteq \mathcal{E}$ is its input event set, and U_h is its ordered sequence of tool calls u_r paired with returned event sets $E_r \subseteq \mathcal{E}$. Events may carry arbitrary, agent-determined payloads. Input edges $e \to h$ record event receipt, while output edges $h \to e'$ record events produced by the activation's calls.

A.2 TAG: Task Activity Graph.

The TAG $\mathcal{T}$ is a directed bipartite graph over the sets of task versions $\mathcal{Q}$ and task actions $\mathcal{A}$, represented as

$$q = (k, \delta, \sigma) \in \mathcal{Q}, \quad \delta = (g, c), \quad a = (h, u) \in \mathcal{A}. \tag{5}$$

Here k is the persistent task identity, g is the goal, σ is the state, and c is an evaluation rule whose result $c(g, \sigma) \in \Omega$ may be a score or statement. Each q is an immutable version of task k. An action $a = (h, u)$ records a task tool call $u \in \mathcal{U}_T$ issued by activation h. Its incoming edges identify existing input versions, and its outgoing edges identify newly created output versions.

The tools in Table 1 create tasks, revise their specifications or states, and split or join work. Open creates a fresh task identity with status open. Edit changes $\delta = (g, c)$, while update changes σ; both create a new version with the input version as its predecessor. Close(k) closes task k and propagates closure backward through its recorded ancestry. At a split, closure propagates to the parent only when all child tasks produced by that split are closed.

A.3 TEA: TAG, Environment, AIG.

TEA couples the AIG $\mathcal{D}$, TAG $\mathcal{T}$, and environment $\mathcal{Z}$ via tool calls and evidence attachments. The tools are

$$\mathcal{U}_\phi = \{\mathrm{send}_h, \mathrm{attach}_h\}, \qquad \phi = (h, x, q) \in \Phi. \tag{6}$$

Each evidence attachment ϕ links its source activation $h \in \mathcal{H}$ to an existing task version $q \in \mathcal{Q}$ through an arbitrary payload x. Evidence remains attached to that version and does not automatically carry over to later revisions. The call $\mathrm{send}_h(e, J)$ sends an existing event $e \in \mathcal{E}$ to agents $J \subseteq \mathcal{I}$, while $\mathrm{attach}_h(\phi)$ adds ϕ to the evidence relation Φ.

Environment Execution. The shared environment Z has application-defined state, transition dynamics, and observation rules. An environment call $u \in \mathcal{U}_Z$ may execute over multiple primitive steps, with duration and outcome dependent on the evolving environment.

B DAG Sufficiency of AIG and TAG

Proof of Proposition 1. An activation's input events already exist when it starts, and its output events are created by its subsequent calls. Every edge therefore connects an existing node to a newly created

node, so adding these edges cannot create a cycle. Each output event has one producing call and hence one parent activation. This argument allows activation intervals to overlap.

Let $G = (V, F)$ be any finite DAG. Create an activation h_v for each vertex v in topological order. Let h_v produce a fresh event e_v, and include e_v in the inputs of h_w exactly when $(v, w) \in F$. Thus, $h_v \to e_v \to h_w$ represents exactly the dependencies in G. Multiple predecessors provide multiple input events, and multiple successors receive the same event. Execution times consistent with this construction are recorded as annotations. □

Proof of Proposition 2. Let $G = (V, F)$ be any finite DAG. Process its vertices in topological order, assigning each vertex v a task version q_v with a fresh task identity. If v has no predecessors, create q_v with OPEN. If v has one predecessor p, apply SPLIT to q_p, creating q_v and one unused auxiliary version, both with fresh identities. If v has multiple predecessors, apply JOIN to exactly their corresponding versions, creating q_v with a fresh identity. These calls are valid because all input versions already exist, historical versions remain reusable, and the join inputs have distinct identities.

Project each path $q \to a \to q'$ to a version dependency $q \to q'$. The subgraph on the designated versions $\{q_v : v \in V\}$ is exactly G: each construction step uses precisely the required predecessors. Every auxiliary version remains a leaf, with one added for each vertex having one predecessor. Applying the calls in construction order gives timestamps consistent with all dependencies. □

C COMPLETE EXPERIMENTAL SETUP

Tasks. Silo-Bench comprises 30 algorithmic task types across three difficulty levels. Each agent receives a local information shard and must produce the required answer through coordination with the team. We evaluate five agents on ten tasks per level. This setting tests whether agents can combine distributed information into correct collective outputs. The code-refactoring suite contains six repositories grouped into easy (`shop`, `plan`), medium (`ledger`, `freight`), and hard (`stay`, `payroll`) tasks. Agents seek to reduce source lines of code (LOC) while preserving behavior as measured by the supplied test suite. The four-condition code ablation is conducted on `shop` alone, with five runs per condition. The six-repository lesson experiment also uses five runs per repository and condition. The Full row of the shop ablation is round 1 of the lesson loop.

Agents and execution. The current experiments use five agents powered by `gpt-5.6-terra`. In the current code experiments, agents can read and edit every source file, with no predefined roles or file assignments in the initial policy. Tests cannot be modified, and test tracebacks are disabled. Each agent retains a five-step memory, and the team shares a maximum budget of 100 agent steps. Code runs can end through `terminate`, which ends the run immediately, or `request_terminate`, which ends it once all agents agree. Silo-Bench runs end once every agent has used `submit`, subject to the applicable execution budget. The older transfer experiment below uses its separately reported protocol.

Environment actions. Table 8 lists the environment-specific tools. All six code repositories expose the same tools. TAG and direct messaging tools are shared collaboration interfaces and are described separately.

Collaboration interfaces. We compare four conditions formed by enabling or disabling TAG and direct messaging: *Full* (TAG and messaging), *TAG only*, *Comm only*, and *Neither*. The messaging ablation removes `send`; the TAG ablation removes all task tools, evidence attachments, and task-related observations. Other prompts, permissions, budgets, and initial information are held fixed apart from the necessary interface changes. TAG-only agents can still exchange information indirectly through the shared task representation. Thus, these conditions compare available coordination interfaces, rather than the presence or absence of all information exchange. In the code environment, the shared repository also remains a potential coordination channel.

Task observations. Agents access the shared task representation through task tools. `observe()` reads only TAG: its default result lists open task identities and the frontier of their unconsumed versions. It omits historical versions, closed tasks, and evidence payloads. Agents can instead call

Table 8: Environment-specific actions for Silo-Bench and code refactoring. Square brackets indicate optional arguments.

Action	Arguments	Description
Code refactoring		
`list_files`	None	Lists all repository files.
`read_file`	`path`	Returns a file's contents and non-blank line count.
`edit_file`	`path`, `old`, `new`	Replaces an exact, uniquely occurring source snippet. Empty replacement text deletes the snippet. Tests cannot be edited.
`measure`	None	Returns non-blank source line counts per module and for the package as a whole.
`run_tests`	[`path`]	Runs the full test suite or a specified test module; returns test counts, failures, errors, and output.
Silo-Bench		
`read_data`	None	Returns the calling agent's local information shard.
`submit`	`answer`	Submits a final answer and ends the agent's participation. The episode ends when all agents have submitted.

`observe(task)` with a task or version identifier to retrieve that identity's full history, including all versions in chronological order, creator and action information, attached evidence payloads, current-version indicators, and split/join provenance. This makes task histories available on demand without placing the complete history in every observation. The same task configuration is the default return of other TAG tools unless another return view is requested. These observations do not directly inspect AIG or the environment; environmental feedback is obtained through the corresponding environment tools. Recorded observation calls produce AIG events even though they do not modify the task graph.

Metrics. We evaluate task outcomes and interaction effort using the following metrics:

- **Test-suite pass rate (Code Refactoring):** The fraction of runs whose final repository passes all supplied tests.
- **LoC reduction (Code Refactoring):** The percentage reduction in non-blank source lines relative to the initial repository, excluding tests.
- **Success rate (SILO-BENCH):** The fraction of tasks for which every agent answers exactly correctly.
- **Exact correctness (S, SILO-BENCH):** The fraction of agents answering exactly correctly, averaged over Silo-Bench tasks.
- **Partial correctness (P, SILO-BENCH):** The benchmark's task-specific partial-credit score, averaged over agents and Silo-Bench tasks.
- **Agent steps:** The total number of agent steps taken by the team, averaged over runs.
- **Tool calls:** The number of direct messaging, TAG, and test calls, reported separately where available.

Code outcomes are measured on the final repository, without selecting an earlier passing checkpoint. The code-ablation table reports LoC reduction over all runs and among passing runs. The corresponding curves include failed runs, with shaded regions indicating 95% confidence intervals. In the six-repository lesson experiment, LoC reduction is averaged over passing runs within each repository; the overall value is the unweighted mean of these repository means.

Steps measure interaction effort, rather than wall-clock time or token cost. Agreement to stop records a coordination outcome and does not imply that the final repository passes the tests.

C.1 Code Refactoring Tasks

We construct six code-refactoring tasks spanning order processing, day planning, bookkeeping, parcel freight, hotel booking, and payroll (Table 9). Each task provides a functionally correct but redundantly written Python package using only the standard library, together with a fixed `unittest` suite that serves as its behavioral specification. The repositories form three size groups, with 11, 20, and 30 modules and approximately 2,200–7,500 non-blank source lines.

Agents collaboratively reduce non-blank source lines while preserving behavior as measured by the supplied tests. In our experiments, five agents share access to all source files and tests, with no initial file assignments; tests cannot be modified. Each run has a shared budget of 100 agent steps. We evaluate the final repository using test-suite pass rate and LoC reduction, reporting reduction among passing runs separately. The interface ablation uses `tinyshop`, while the lesson experiments cover all six repositories.

Table 9: Code-refactoring repositories grouped by size. Module counts include the package's `__init__` module. LoC counts initial non-blank source lines, excluding tests.

Size	Repository	Domain	Modules	LoC	Test files	Tests
Easy	`shop`	Order processing	11	2,219	10	62
	`plan`	Day planning	11	2,380	10	69
Medium	`ledger`	Bookkeeping	20	5,082	19	143
	`freight`	Parcel freight	20	5,161	19	154
Hard	`stay`	Hotel booking	30	7,484	29	213
	`payroll`	Payroll	30	7,399	29	214

D Extended Results

D.1 Collaboration Policy Discovery/Recovery

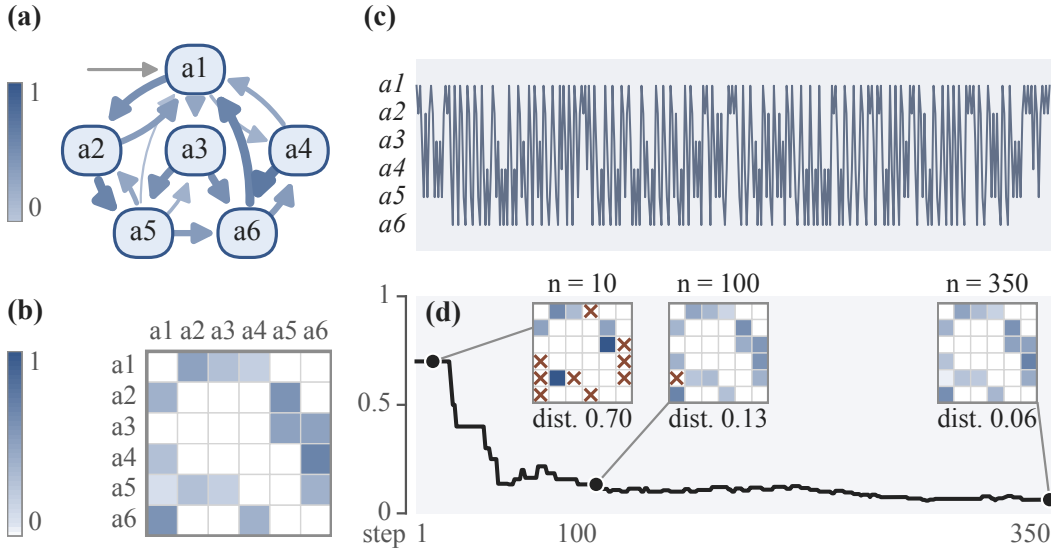


Figure 8: Routing recovery: (a) routing; (b) its weighted adjacency matrix of edge probabilities; (c) one execution's $\mathcal{D}$; (d) its L_∞ distance to (b) as it runs. Squares: the matrix of edge frequencies recovered from (c) after n steps; crosses: edges not yet run.

D.2 Communication Patterns under Ablation

Figure 9 aggregates each record into who sent to whom over the whole episode. The two right-hand panels are empty by construction: with `send` removed, there is no edge to draw.

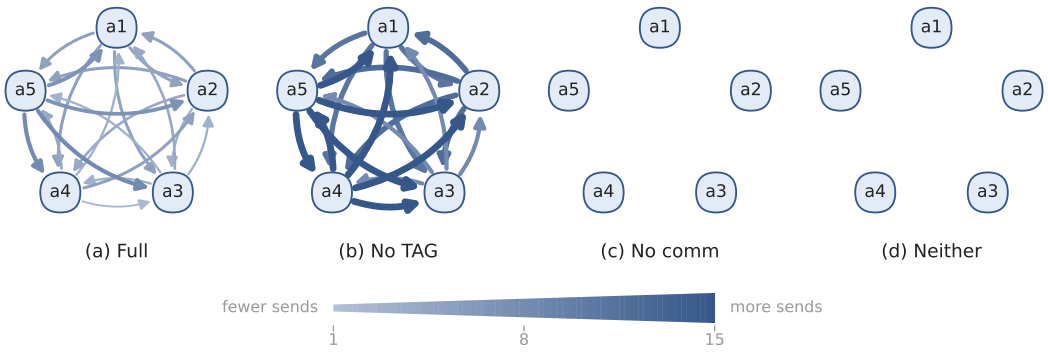
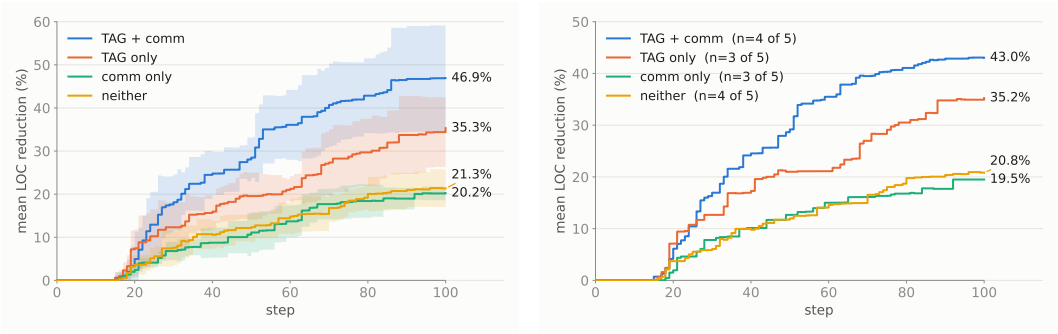


Figure 9: Interaction graphs of the five agents, summed over five seeds per setting: an edge $X \to Y$ indicates that an activation of X sent an event to Y (a broadcast counts once per recipient); a darker, thicker edge carried more events, on one scale for all panels. Removing TAG raises traffic rather than lowering it: 244 events over 20 edges without it against 94 over 20 with it, and it does so in every seed (40–56 events per episode against 8–35). Panels (c) and (d) are empty by construction: with `send` removed there is no edge to draw.

D.3 Ablations on Clean Runs

The main text reports the mean LOC reduction over all five runs of each setting (Figure 7). A run can remove lines and still end with a failing test suite, so here we also report the same curves over the *clean* runs only: those whose suite passed when the agents stopped. Figure 10 places the two side by side. All runs use the same package (shop), five agents with no owned files, a budget of 100 steps, and seeds 0–4; a run that stopped before the budget keeps its final size for the remaining steps. The clean runs are 4 of 5 for TAG + comm, 3 for TAG only, 3 for comm only, and 4 for neither.



(a) All runs ($n = 5$); shaded: ± 1 standard deviation

(b) Clean runs only.

Figure 10: Mean LOC reduction by step for each setting. Final reductions: (a) 46.9%, 35.3%, 21.3%, 20.2%; (b) 43.0%, 35.2%, 20.8%, 19.5%, for TAG + comm, TAG only, neither, and comm only.

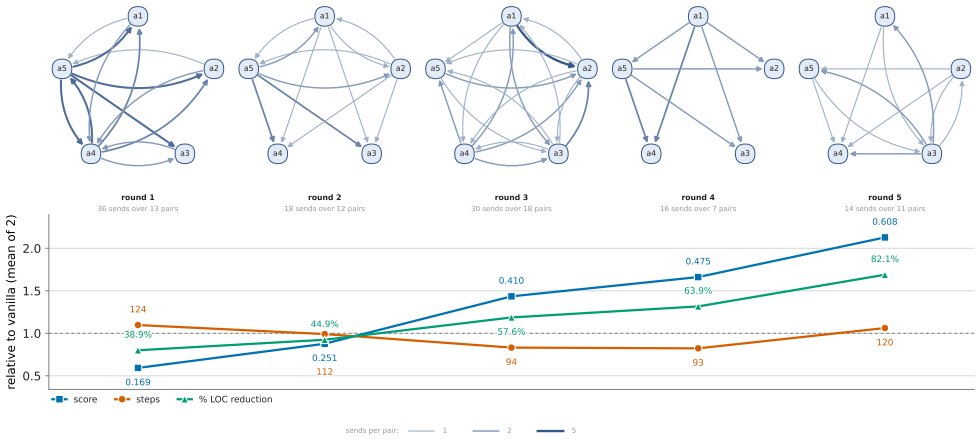


Figure 11: The five rounds from agent views: how the team's interaction changed as each round inherited the previous round's notes.

D.4 INFUSE: ITERATIVE IMPROVEMENT EXAMPLE

D.5 EXAMPLE TRACES

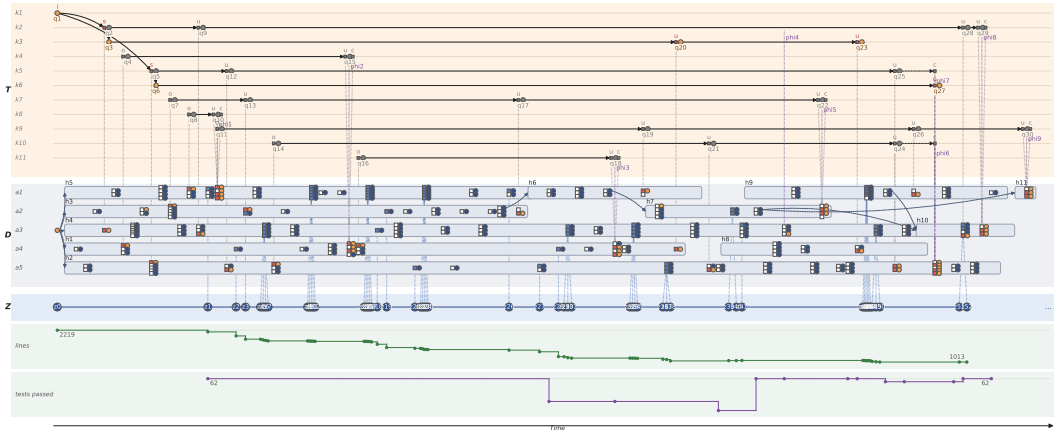


Figure 12: Episode record, full condition: TAG and communication both available.

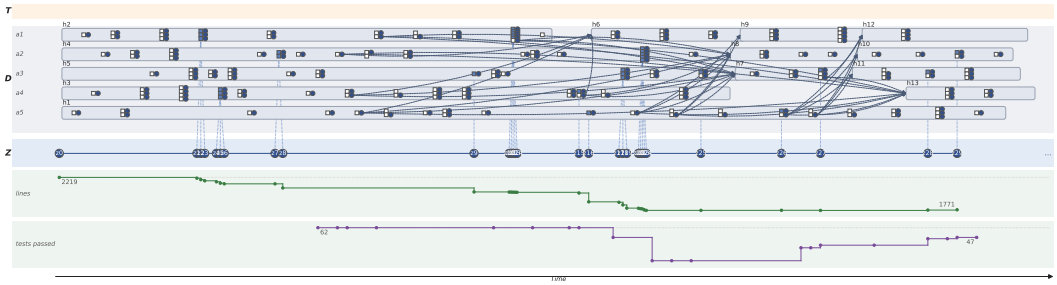


Figure 13: Episode record without TAG: the eight task tools removed.

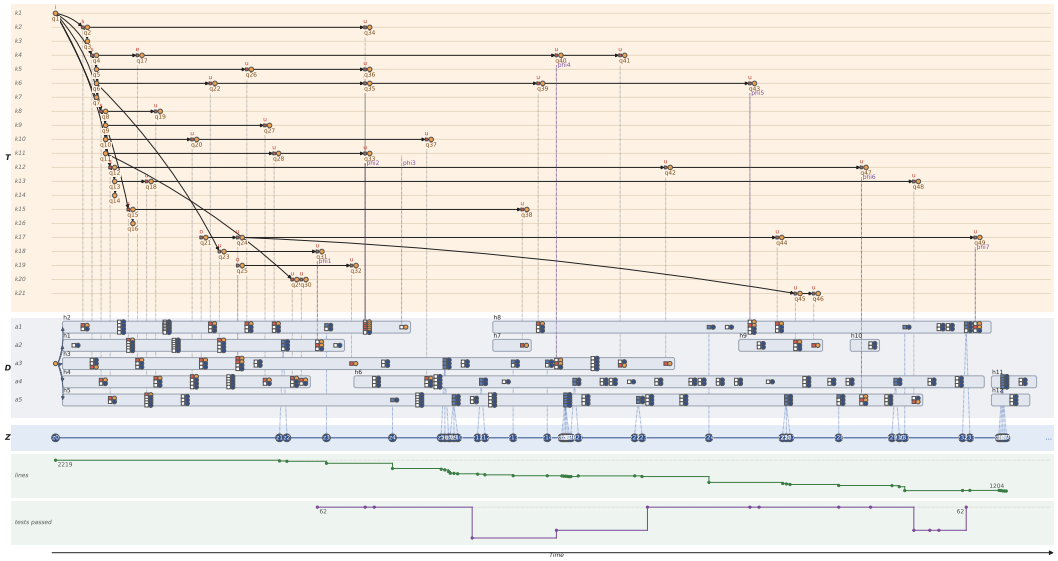


Figure 14: Episode record without communication: `send` removed.

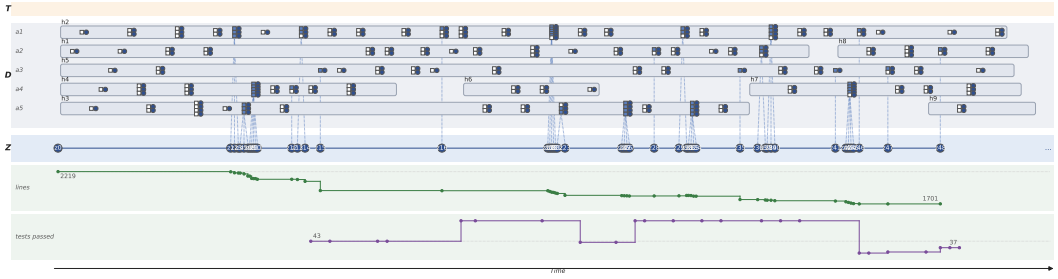


Figure 15: Episode record with neither TAG nor communication: the agents share the repository and nothing else.

D.6 REPRESENTATIVE SUCCESSFUL RUNS

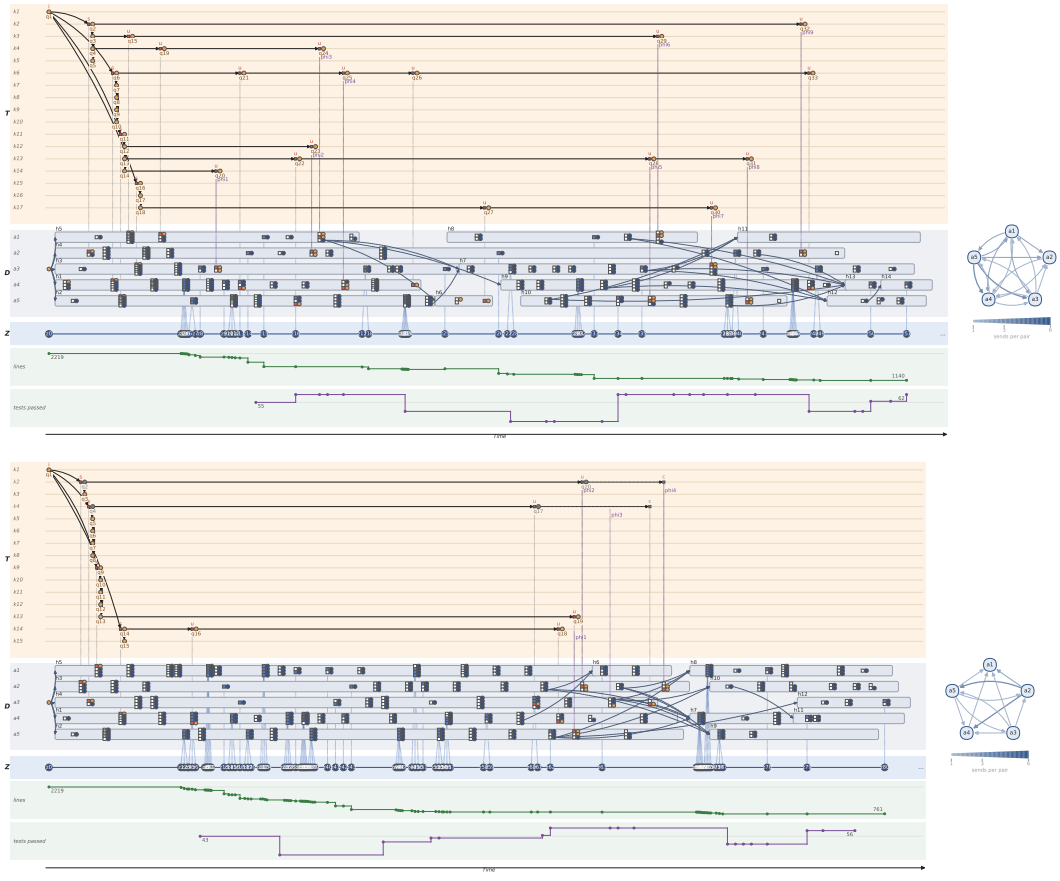


Figure 16: `shop` (11 modules). Top: round 1; bottom: round 2, played with the lessons distilled from round 1. Each row is one episode's record: T, the task graph; D, the five agents' activations and the sends between them; Z, the repository's transitions, with the package's non-blank lines and the tests passed beneath. On the right is its agent view: an edge $X \to Y$ when X sent an event to Y, darker and thicker for more sends (1 to 6 per pair, one scale in every code figure of this appendix). Round 1 trims the package from 2219 to 1140 lines (48.6%) with all 62 tests passing; round 2 trims it to 761 (65.7%), still passing.

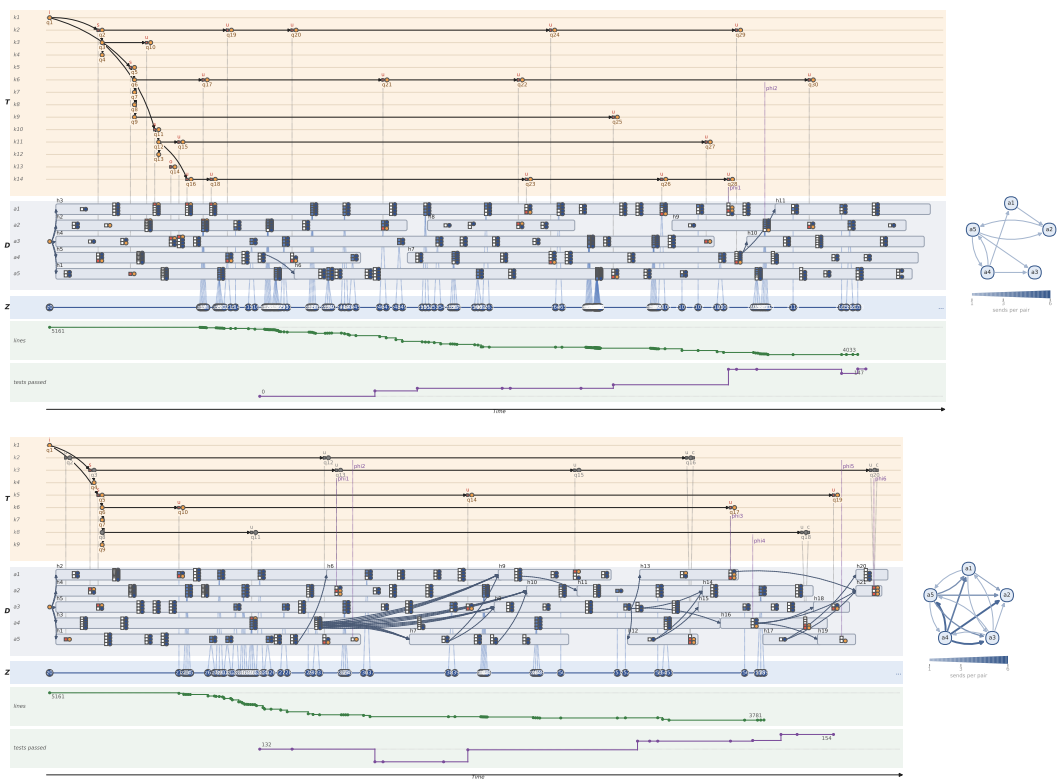


Figure 17: `freight` (20 modules), seed 4; layout as in Figure 16. Round 1 trims 5161 to 4033 lines (21.9%) but leaves 7 of 154 tests failing; round 2 trims to 3781 (26.7%) with all 154 passing, in 92 of its 100 steps.

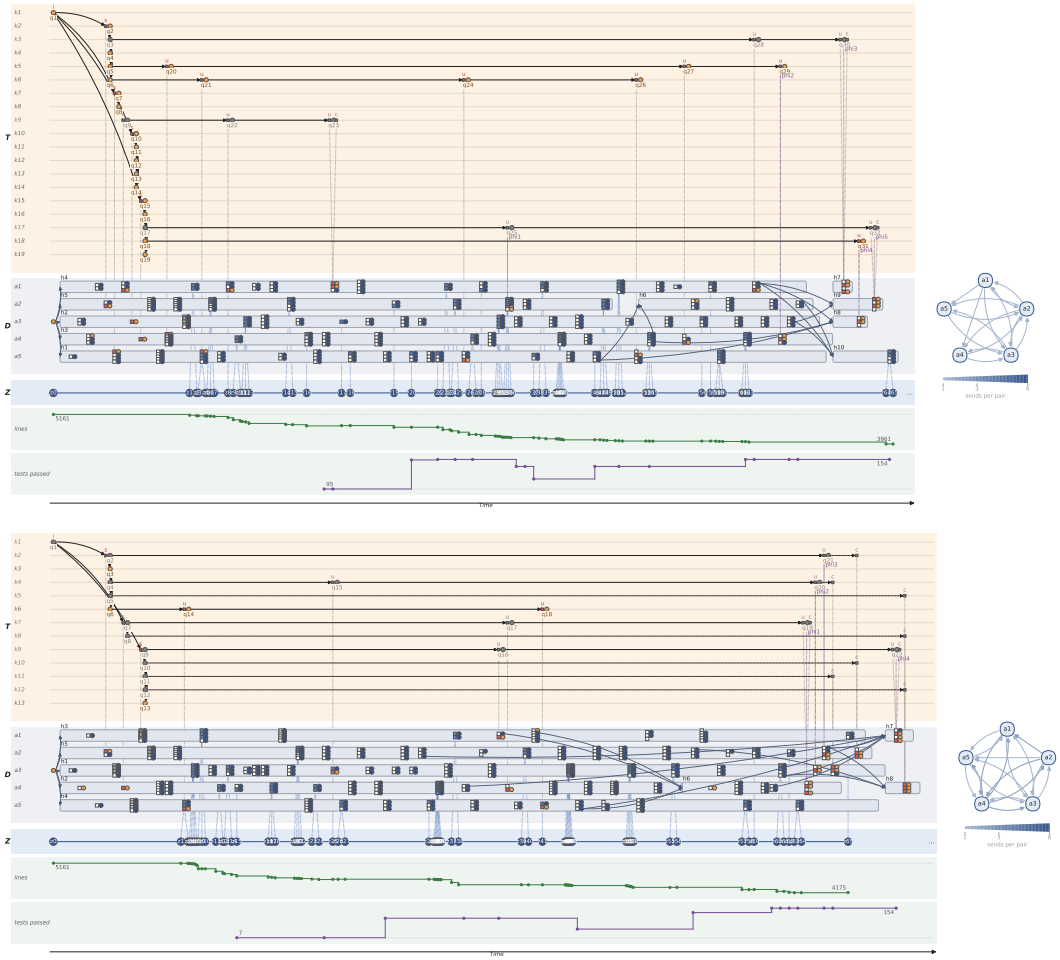


Figure 18: `freight`, seed 3; layout as in Figure 16. Both rounds keep all 154 tests passing; round 2 trims 5161 to 4175 lines (19.1%), less than the 3961 (23.3%) of round 1.

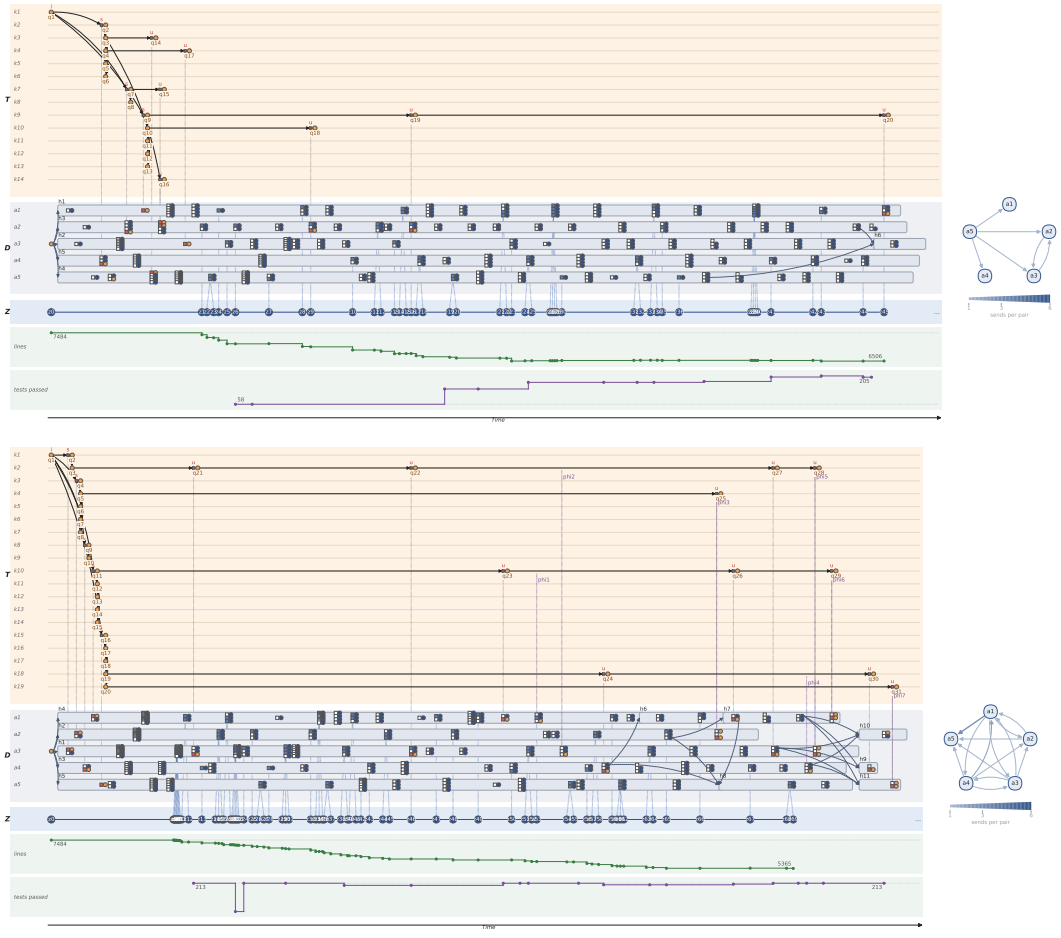


Figure 19: `stay` (30 modules), seed 1; layout as in Figure 16. Round 1 trims 7484 to 6506 lines (13.1%) and leaves the suite red (205 of the 206 tests run pass); round 2 trims to 5365 (28.3%) with all 213 tests passing.

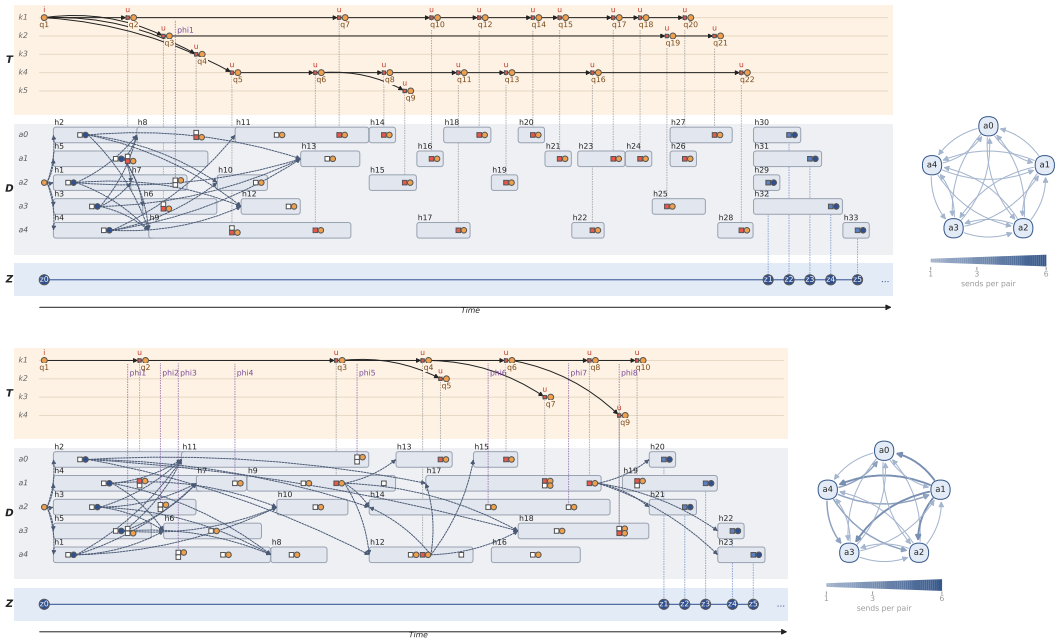


Figure 20: Silo-Bench I-10 (Standard Deviation, level I), full condition. Top: round 1; bottom: round 2, played with the lessons distilled from round 1 on this task. Layout as in Figure 16, without the two strips under Z: each of the five agents reads its own shard of the data and submits one answer. The agent views share one scale in every Silo-Bench figure of this appendix (1 to 6 sends per pair). Round 1 fails ($S = 0$) after 78 steps; round 2 succeeds ($S = 1$) in 50.

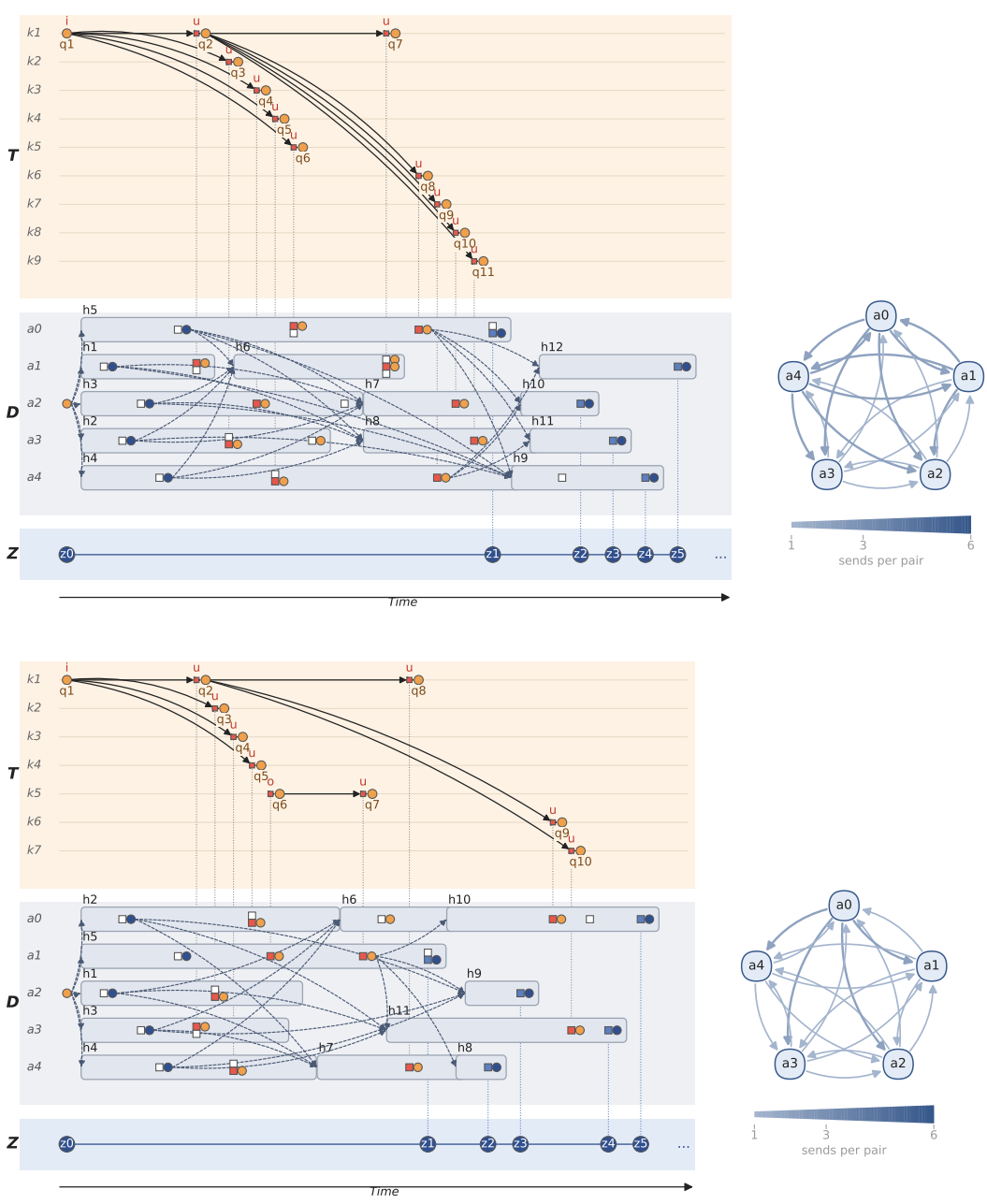


Figure 21: Silo-Bench I-07 (Average Value, level I); layout as in Figure 20. Both rounds succeed ($S = 1$), in 23 and 25 steps.

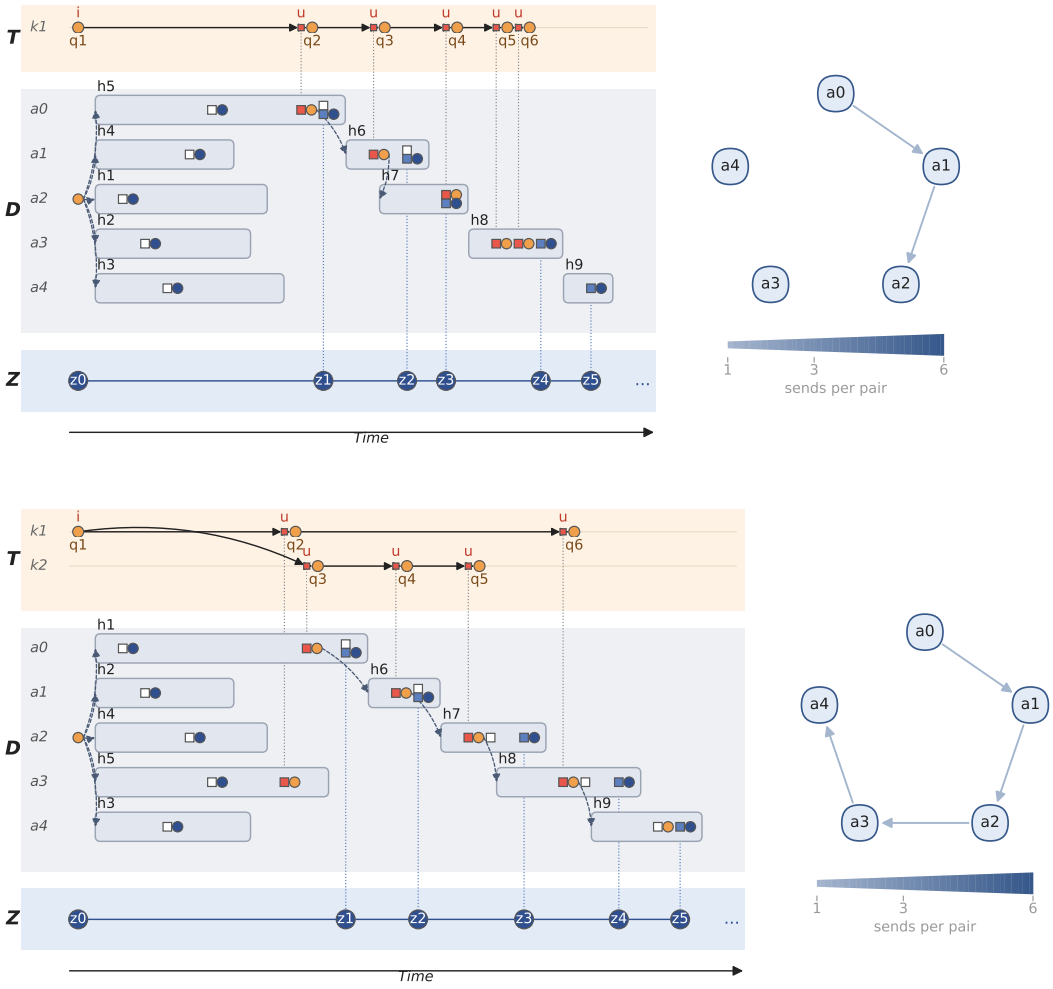


Figure 22: Silo-Bench II-20 (Pipeline Hash, level II); layout as in Figure 20. Round 1 scores $S = 0.40$, its relay breaking off after a_2; round 2 completes the chain $a_0 \to a_1 \to a_2 \to a_3 \to a_4$ and succeeds ($S = 1$).

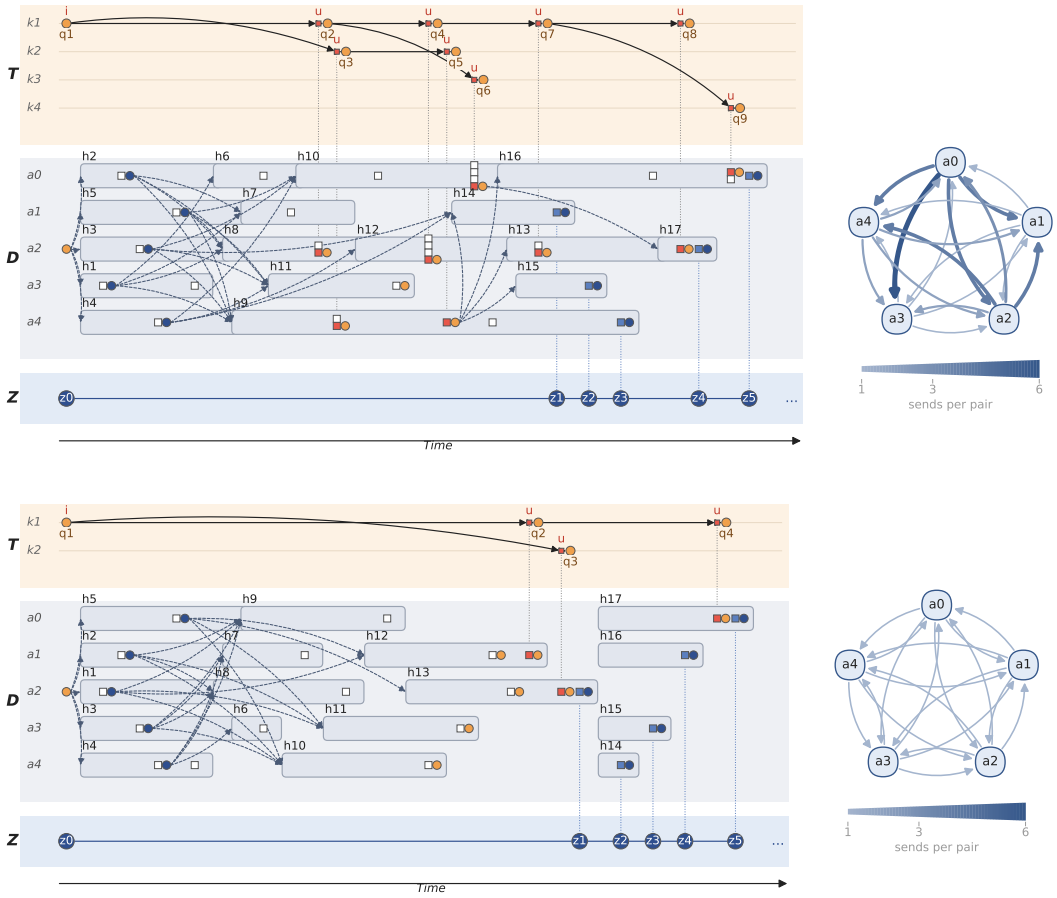


Figure 23: Silo-Bench III-24 (BFS Distance, level III); layout as in Figure 20. Both rounds succeed ($S = 1$); round 2 needs 22 steps against 26, 5 sends against 16, and 3 task actions against 8.