NANOMA: HOW DOES AGENT REACT ENABLE FLEXIBLE MULTI-AGENT TOPOLOGIES?

Anonymous authors
Paper under double-blind review

ABSTRACT

Long-horizon tasks continually reveal new dependencies and evidence, requiring multi-agent collaboration to adapt during execution. Central planning concentrates coordination in one controller, while recursive delegation distributes task decomposition but offers limited support for revising existing cross-branch collaboration. We introduce **NanoMA**, a multi-agent ReAct (Reasoning + Acting) framework that treats collaboration structure as an executable and dynamically revisable component of problem solving. Within each agent's ReAct loop, execution actions are combined with four topology actions—`spawn`, `kill`, `query`, and `send`—for creating, terminating, observing, and contacting agents. Execution traces show agents combining these actions in response to task feedback to form diverse collaboration structures, rather than following predefined topology templates. On RepoZero-Py2JS Hard with GPT-5.6 Luna High, NanoMA achieves a 66.91% all-pass rate, exceeding the best-performing baseline in our comparison by 13.67 percentage points. On EdgeBench, it obtains the highest overall score among the evaluated frameworks with both GPT-5.6 Luna and Qwen3.7-Plus. Communication ablations show higher task accuracy with cross-branch query access and better performance with targeted messages than with broadcasting. Code is available anonymously at `https://anonymous.4open.science/r/NanoMA-private-2CC6/`.

1 INTRODUCTION

As large language model (LLM) agents move beyond single-turn question answering to long-horizon tasks, systems such as Claude Code and Codex have introduced multi-agent collaboration for tool use, multi-file work, and plan revision (Anthropic, 2026; OpenAI, 2026). Benchmarks such as EdgeBench evaluate sustained interaction with environments and adaptation to feedback (Zhu et al., 2026). In these settings, an initial task decomposition is insufficient. New dependencies emerge, evidence uncovered in one branch redirects work in another, and some subtasks cease to be useful. Effective collaboration requires agents to revise assignments, exchange information, and terminate unproductive work as execution unfolds. The challenge is how to adapt the collaboration structure to evolving demands.

Central planning addresses coordination through a controller that decomposes tasks, coordinates execution, and integrates results, as exemplified by Magentic-One, Evolving Orchestration, and AOrchestra (Fourney et al., 2024; Dang et al., 2025; Ruan et al., 2026) (Figure 1(a)). This organization helps maintain the overall goal and allows plans to change with feedback. However, it also concentrates state integration and coordination decisions in one node. As parallel branches and intermediate results accumulate, the planner faces a growing coordination load. Local evidence can lose detail when summarized and relayed, while changes to assignments require another exchange with the planner. Prior work identifies these effects as communication and integration bottlenecks of centralized coordination (Mao & Mirhoseini, 2026).

Node-recursive methods, including ReDel, Recursive Agent Optimization, and ROMA, distribute task decomposition and delegation across execution nodes, allowing subagents to create and solve further subtasks (Zhu et al., 2024; Gandhi et al., 2026; Alzu'bi et al., 2026) (Figure 1(b)). This supports local divide-and-conquer, but recursive expansion alone does not specify how existing collaboration should change. ROMA, for example, supports dependencies among sibling subtasks while

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.

retaining a common plan-execute-aggregate loop (Alzu'bi et al., 2026). The relevant distinction is therefore not whether execution takes the form of a graph, but whether agents can revise that graph as execution unfolds by establishing cross-branch links, reorganizing ongoing work, and terminating branches that are no longer useful.

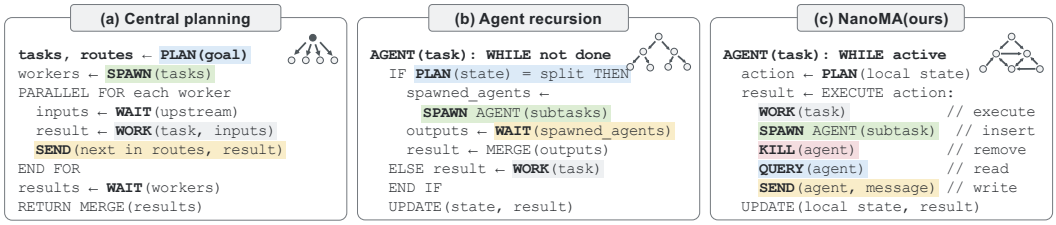


Figure 1: Schematic comparison of multi-agent orchestration. (a) A planner creates workers and routes result exchanges. (b) Each node chooses local work or delegation, then waits for child results before updating. (c) Agents select execution or topology actions, including spawning, terminating, observing, and communicating, allowing collaboration to evolve beyond the provenance tree.

To address this challenge, we introduce NanoMA, a multi-agent framework that gives each agent autonomy over both task execution and collaboration. NanoMA separates task provenance from the evolving needs of collaboration. An agent's task origin is fixed when it is created, whereas the information and collaborators it requires may change throughout execution. Accordingly, NanoMA distinguishes between creation links, which record task origins, and collaboration links, which can be established across levels and branches as needed (Figure 1(c)). Each agent acts as both planner and executor, combining local progress, environmental feedback, and evidence from other agents to decide whether to explore further, revise its collaborations, or terminate unproductive work.

To operationalize this autonomy, NanoMA equips every active agent with four topology actions. These actions provide basic operations over the collaboration structure: `spawn` creates a new agent node, `kill` terminates an ongoing branch, `query` retrieves the state of other agents, and `send` communicates information or instructions to them. Together with the execution action `work`, they extend ReAct's interleaved reasoning, action, and observation (Yao et al., 2023) to both task execution and topology control. This action space allows each agent to advance its assigned task while revising the execution graph in response to newly discovered dependencies, evidence, and state changes. Creation links preserve task provenance, whereas information flow and collaboration links can evolve throughout execution to accommodate emerging needs (Figure 2). The main contributions are as follows:

- We introduce NanoMA, a multi-agent framework that distributes planning and execution across agents while separating task provenance from collaboration structure.
- We introduce four topology actions, `spawn`, `kill`, `query`, and `send`, which extend agent control from task execution to dynamic collaboration.
- We evaluate NanoMA on RepoZero-Py2JS Hard (Zhang et al., 2026) and EdgeBench (Zhu et al., 2026). Our experiments compare performance across models and orchestration frameworks, analyze emergent collaboration structures, and quantify how query availability and access scope affect correctness and resource use.

2 Related Work

Central Planning in Multi-Agent Systems. Central planning assigns task decomposition and coordination to a controller that routes work to specialist agents and aggregates their outputs. Magentic-One, AgentOrchestra, Evolving Orchestration, and AOrchestra instantiate or extend this pattern through explicit orchestration, workflow adaptation, and subagent creation (Fourney et al., 2024; Zhang et al., 2025; Dang et al., 2025; Ruan et al., 2026). These systems provide clear global objectives and responsibility boundaries, but concentrate state integration and replanning at the controller, which can create communication bottlenecks and information loss during aggregation (Mao & Mirhoseini, 2026; Rizvi-Martel et al., 2025). NanoMA instead distributes topology decisions across the participating agents.

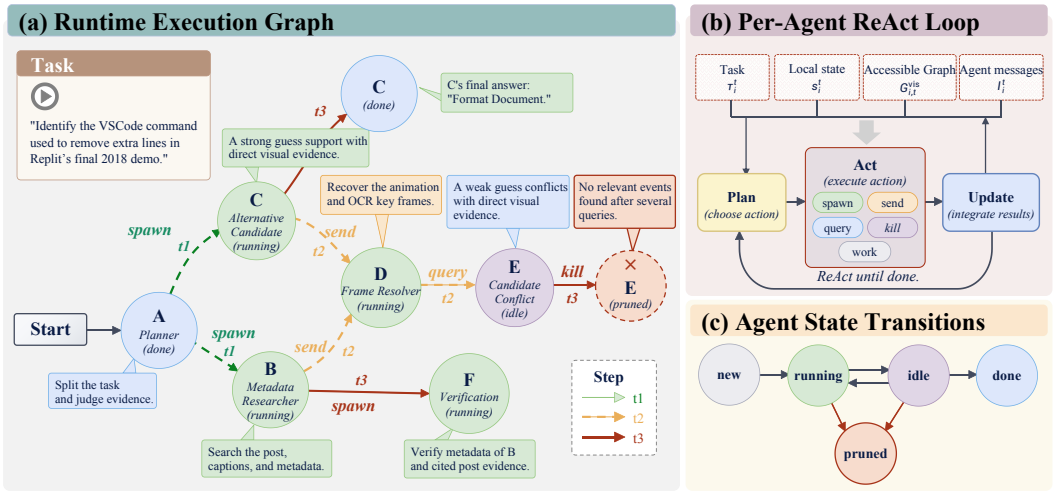


Figure 2: Overview of NanoMA's dynamic execution graph and agent control loop. (a) Three non-consecutive snapshots (t_1, t_2, t_3) show agents expanding collaboration via `spawn`, exchanging information via `send` and `query`, and pruning an unproductive branch via `kill`. Colored action edges record collaboration events. (b) Per-agent ReAct loop: each agent uses its task, local state, accessible execution graph, and incoming messages to choose `work` or a topology action, then updates its state before continuing or terminating. (c) Agent-state transitions among *new*, *running*, *idle*, *done*, and *pruned*. Both *running* and *idle* can transition to *done*; entering *idle* is not required for completion.

Node Recursion in Multi-Agent Systems. Recursive systems allow an agent to decompose a local assignment and delegate subtasks to descendant agents, enabling divide-and-conquer without a separate global scheduler. ReDel provides a toolkit for recursive multi-agent systems, while Recursive Agent Optimization, ROMA, and Recursive Agent Harnesses investigate recursive decomposition and long-horizon execution (Zhu et al., 2024; Gandhi et al., 2026; Alzu'bi et al., 2026; Lumer et al., 2026). Omnigent's agent specification adopts the same node-level delegation setting (Omnigent, 2026). These systems broaden local planning, but their control interfaces are primarily expressed through recursive delegation and parent–child aggregation. Consequently, they provide less support for reorganizing cross-branch roles or selectively shrinking them during execution. NanoMA addresses this complementary gap by separating provenance from collaboration and exposing topology actions for cross-lineage observation, communication, expansion, and termination.

3 Method

3.1 Overview

NanoMA starts each task with a single agent and dynamically expands the collaboration structure during execution. Figure 2(a) shows the resulting multi-agent execution graph $G_t = (V_t, E_t^{\text{prov}}, E_t^{\text{comm}})$, where V_t contains agents created by time t, E_t^{prov} records creation links, and E_t^{comm} records directed communication links from addressed messages. Each agent follows the ReAct loop in Figure 2(b), selecting execution or topology actions based on its context and incoming information. Its lifecycle state ℓ_i^t, shown in Figure 2(c), tracks whether it is running, waiting, completed, or pruned. Together, these decisions and state transitions drive graph evolution and enable task-level orchestration.

3.2 Runtime Execution Graph

Figure 2(a) illustrates this process for a task in which NanoMA must identify the VSCode command shown in a video. We highlight three representative moments, t_1, t_2, t_3, ordered in time but not

necessarily corresponding to consecutive execution steps; additional actions and structural changes may occur between them.

At t_1, the initial planning agent A invokes `spawn` to create B and C, assigning them to metadata retrieval and video-frame analysis. B and C are added to V_t, together with the provenance links (A, B) and (A, C) in E_t^{prov}. NanoMA also creates E to pursue an independent line of reasoning in parallel with C and generate a separate candidate answer.

The execution then introduces D, a frame-resolution agent, to take over the findings produced by earlier branches and continue integrating evidence. At t_2, D uses `query` to read E's evidence, while B and C finish their subtasks and `send` their results to D. These messages add (B, D) and (C, D) to E_t^{comm}. D can therefore combine evidence it actively retrieves with results delivered by other branches, rather than relying on a single predefined information-flow path.

At t_3, D revises its judgment after receiving C's message and rejects E's evidence. NanoMA subsequently terminates E's branch through `kill`. Although E no longer participates in active execution, its historical record remains in V_t. Meanwhile, B creates the verification agent F, adding F to V_t and (B, F) to E_t^{prov}. After transmitting its result and completing its own task, C transitions to $\ell_C^{t_3} = \texttt{done}$, represented as *C (done)* in the graph. This example shows how NanoMA can introduce verification when needed while pruning branches that are complete or no longer useful.

These dynamics are not imposed by a fixed sequence of global stages. Instead, within NanoMA's multi-agent ReAct loop (Yao et al., 2023), each agent selects both execution actions, such as `work`, and topology actions based on its goal, available evidence, and feedback from the environment. The resulting local decisions continuously construct and revise the global execution graph G_t, allowing NanoMA's collaboration structure to adapt as new evidence emerges and task requirements change.

3.3 Per-Agent ReAct Loop

Figure 2(b) illustrates NanoMA's multi-agent ReAct loop. At each step, the LLM reasons over the current task and available evidence, selects an action, and incorporates the resulting observation to refine its plan. The action space comprises *topology actions*, including `spawn`, `kill`, `query`, and `send`, for creating, terminating, inspecting, and communicating with agents, as well as *execution actions* that advance the task using task-specific tools.

Let s_i^t denote agent i's local state and $G_{i,t}^{\text{vis}}$ its accessible execution-graph information. The cycle is $(s_i^t, G_{i,t}^{\text{vis}}) \longrightarrow a_i^t \longrightarrow (s_i^{t+1}, G_{i,t+1}^{\text{vis}}) \longrightarrow a_i^{t+1}$. Action outcomes update the prompt context through changes to s_i^t, while topology-changing actions also modify the active agents or communication links in G_t. Appendix A.3 details the state and graph effects of individual tools. Local state includes history h_i^t, resource budget b_i^t, inbox I_i^t, and lifecycle state ℓ_i^t, defined fully in Section 3.4. The four topology actions correspond to insertion, deletion, lookup, and update in data structures (Cormen et al., 2022). Each description below specifies the direct effect of one successful call.

• **`spawn`.** An agent creates another agent for a subtask, adding it to V_t and recording its creation link in E_t^{prov}. The new agent has its context and local plan, with the same execution and topology actions. Combining `spawn` with `wait` expresses recursive delegation, while continuing without waiting allows parent and child to work in parallel. Admission details appear in Appendix A.2.

• **`kill`.** An agent terminates itself or a selected descendant whose work is redundant, obsolete, or no longer needed. The target leaves active execution but remains in V_t, with its history and existing results retained. The graph can therefore contract without erasing previously collected evidence; the exact termination scope is given in Appendix A.5.2.

• **`query`.** An agent reads the state, progress, and available results of an accessible agent. If a query returns observation o_i^{t+1}, the caller records $h_i^{t+1} = h_i^t \oplus (a_i^t, o_i^{t+1})$, where $\oplus$ incorporates new records into the relevant history or inbox. The call consumes part of b_i^t, but does not modify the queried agent or add or remove nodes or edges in G_t. It supplies evidence when needed, without copying the full context in advance or asking the target to generate a fresh reply.

• **`send`.** An agent sends evidence, constraints, progress, or a suggested adjustment to a specified recipient. For a message m_{ij}^t from i to j, $I_j^{t+1} = I_j^t \oplus m_{ij}^t$ and $E_{t+1}^{\text{comm}} = E_t^{\text{comm}} \cup \{(i, j)\}$. The sender records the action and its return value r_i^{t+1} through $h_i^{t+1} = h_i^t \oplus (a_i^t, r_i^{t+1})$. The recipient uses

the message in later decisions; if idle, it can be awakened through ℓ_j : `idle` $\rightarrow$ `running`. Thus, `send` changes the recipient's available information and communication links, rather than deciding its next action. Message handling is detailed in Appendix A.4.

Together, these actions support two complementary communication modes. The `query` action provides on-demand access, whereas `send` actively delivers information to another agent. Because agents may not reliably anticipate the system-wide effects of these choices, NanoMA makes the access scope configurable and evaluates its impact in Section 4.3.1.

3.4 Agent State Transitions

An agent selects its state from task progress, dependencies, incoming information, and remaining resources. Its local state is $s_i^t = (\tau_i^t, h_i^t, Q_i^t, W_i^t, \ell_i^t, b_i^t, O_i^t, I_i^t)$: the current task, effective interaction history, dynamic local task queue, workspace state, lifecycle state, resource state, its own results and artifacts, and unprocessed addressed messages, respectively. In Figure 2(c), `running` denotes execution, `idle` a pause, `done` completion, and `pruned` termination through `kill`.

The decision criterion for yielding is $Q_i^t \neq \varnothing \wedge \mathrm{Ready}(Q_i^t, I_i^t) = \varnothing \wedge b_i^t > 0$. Here, Ready identifies work that can proceed given the dependencies and messages, and $b_i^t > 0$ indicates available budget. The task is unfinished but temporarily blocked. A later addressed message can wake the agent to reassess the available work.

The criterion for normal completion is $Q_i^t = \varnothing \wedge \mathrm{Valid}(\tau_i^t, h_i^t, W_i^t, O_i^t) = 1$. Valid assesses completion from the task, history, workspace, and outputs. An empty queue together with this assessment distinguishes finished work from having nothing to do temporarily.

`idle` pauses blocked work, `done` closes completed work, and `kill` terminates unproductive branches. With finite task expansion and unfinished work eventually resolved or terminated, these local transitions eventually leave no active agents in the graph. The full conditions and argument appear in Appendix A.5.3. The mapping to implemented transitions appears in Appendix A.5.

4 Experiments

The experiments compare agent harnesses across three model backbones: GPT-5.6 Luna High (OpenAI, 2026), Claude Sonnet 5 (Anthropic, 2026), and Qwen3.7-Plus (Alibaba Cloud, 2026). Five organizational patterns are evaluated. OpenHands-bash (Wang et al., 2025) serves as the single-agent baseline for RepoZero-Py2JS, while Codex (OpenAI, 2026) serves as the single-agent baseline for EdgeBench. CrewAI's sequential process (CrewAI, 2026) represents sequential organization. CC-Workflow, instantiated from Claude Code's dynamic workflows (Anthropic, 2026), represents central planning. Omnigent (Omnigent, 2026) represents node recursion, and NanoMA represents node-autonomous organization. Comparisons are performed within each model backbone to avoid attributing model differences to orchestration. The evaluation spans two benchmarks with distinct task characteristics. RepoZero-Py2JS Hard focuses on challenging repository-level tasks that require coordinated changes across files and subsequent execution-based validation, while EdgeBench (Zhu et al., 2026) covers heterogeneous, long-horizon tasks that demand sustained interaction with tools and execution environments.

4.1 RepoZero-Py2JS Hard

Task and metrics. RepoZero evaluates repository generation from scratch through executable black-box verification: an agent must reproduce the behavior of a source repository under cross-language constraints (Zhang et al., 2026). In the Py2JS setting, the source behavior is implemented in Python and the target repository must be implemented in JavaScript. The Hard-split evaluation snapshot contains 139 tasks and 16,323 hidden test cases. The evaluation reports *all-pass rate*, which counts a task as solved only when all of its hidden tests pass, and *micro-averaged hidden-test pass rate* (Micro), which aggregates hidden cases globally to measure functional coverage. Framework comparisons are made within each backbone so that model differences are not attributed to orchestration. Resource use for all three backbones is reported separately in Appendix C.

Table 1: Results on RepoZero-Py2JS Hard. Micro is the reported micro-averaged hidden-test pass rate. Harness type identifies the evaluated organizational pattern. Bold indicates the highest score within each backbone and metric.

Agent Framework	Harness Type	GPT-5.6 Luna High		Claude Sonnet 5		Qwen3.7-Plus	
		All-pass	Micro	All-pass	Micro	All-pass	Micro
OpenHands-bash	Single agent	44.83%	80.03%	43.52%	79.82%	25.899%	57.097%
CrewAI	Sequential	53.24%	75.02%	36.69%	77.11%	28.777%	67.102%
CC-Workflow	Central planning	48.20%	74.69%	24.46%	52.89%	24.460%	54.347%
Omnigent	Node recursive	47.48%	79.20%	48.20%	87.17%	**39.568%**	72.260%
NanoMA	**Node autonomous**	**66.91%**	**93.32%**	**51.80%**	**87.30%**	35.971%	**76.561%**

Results. Table 1 shows that, under GPT-5.6 Luna High, NanoMA achieves the highest all-pass rate, solving 93 of 139 tasks (66.91%). This is 13.67 percentage points above the strongest baseline by all-pass rate, CrewAI. NanoMA also achieves the highest Micro score at 93.32%, exceeding OpenHands-bash by 13.29 percentage points. Under Claude Sonnet 5, NanoMA achieves the highest all-pass rate and Micro score among the evaluated frameworks. Under Qwen3.7-Plus, Omnigent achieves the highest all-pass rate at 39.568%, while NanoMA achieves the highest Micro score at 76.561%.

4.2 EDGEBENCH

Task and metrics. EdgeBench evaluates agents on 51 public tasks spanning six categories: Scientific and Machine Learning (ML), Systems and Software Engineering (SE), Optimization, Knowledge, Formal, and Games. Total and category scores are computed using EdgeBench's normalized scoring protocol. Each task is allotted a two-hour execution budget.

Results. Table 2 compares NanoMA with Codex and the other evaluated frameworks within each model backbone. Under GPT-5.6 Luna, NanoMA achieves the highest total score at 22.1010, surpassing the matched Codex baseline by 2.0570 points. Under Qwen3.7-Plus, NanoMA also outperforms Codex, increasing the total score from 17.516 to 18.581, a gain of 1.065 points. Its category-level strengths vary across backbones. NanoMA leads on Systems & SE, Optimization, and Knowledge under GPT-5.6 Luna, and on Systems & SE, Optimization, Knowledge, and Games under Qwen3.7-Plus. These results are obtained under a fixed two-hour budget per task.

Table 2: Two-hour results on the EdgeBench public-51 benchmark. Scores follow EdgeBench's normalized protocol. Harness type denotes the evaluated organizational pattern. For each model, the highest score in each column is shown in bold.

Model	Agent	Harness Type	Overall Score	Scientific & ML	Systems & SE	Optimization	Knowledge	Formal	Games
GPT-5.6 Luna									
	Codex baseline	Single agent	20.04	22.37	29.51	22.46	11.06	8.82	**15.88**
	CrewAI	Sequential	19.36	39.76	31.94	18.74	12.75	**11.21**	11.58
	CC-Workflow	Central planning	16.60	**42.75**	23.42	18.17	14.65	4.29	15.87
	Omnigent	Node recursive	14.31	19.72	28.14	14.10	13.08	1.04	5.14
	NanoMA	**Node autonomous**	**22.10**	19.46	**34.89**	**23.54**	**25.81**	7.02	14.78
Qwen3.7-Plus									
	Codex baseline	Single agent	17.52	20.79	26.72	18.49	18.90	1.24	15.83
	CrewAI	Sequential	13.09	**35.16**	20.76	13.99	15.98	**1.46**	10.26
	CC-Workflow	Central planning	10.95	14.24	20.17	8.77	12.14	0.75	9.20
	Omnigent	Node recursive	13.06	15.02	22.92	11.95	12.77	0.21	9.06
	NanoMA	**Node autonomous**	**18.58**	24.11	**26.97**	**20.29**	**19.03**	1.18	**17.20**

4.3 Ablation Studies

4.3.1 Query Availability and Scope

Table 3(a) compares three query policies on the 139 RepoZero-Py2JS Hard tasks using GPT-5.6 Luna. Full query clearly outperforms the two restricted settings, achieving a 66.91% all-pass rate and a 93.32% Micro score. Both Direct-family query and query-disabled obtain an all-pass rate of 47.48%, while their Micro scores are 76.13% and 78.77%, respectively. Thus, restricting query access does not improve task-level success, and disabling query slightly improves aggregate hidden-test coverage relative to Direct-family query.

Full query also incurs the highest interaction cost, with 3.53 million (M) mean tokens and 13.23 queries per task. Direct-family query reduces these costs to 2.56M tokens and 5.91 queries per task, but provides no all-pass improvement over query-disabled. These results indicate that cross-branch access is valuable when agents need information from outside their local family. The benefit comes with additional interaction and token costs, suggesting a trade-off between broader information access and execution efficiency.

4.3.2 Send Recipient Scope

Table 3(b) compares one-to-one send, send-disabled, and one-to-all send on EdgeBench using GPT-5.6 Luna. One-to-one send achieves the highest mean score at 22.1010, improving over send-disabled by 5.3142 points and over one-to-all send by 7.4606 points. Disabling send reduces the mean score to 16.7868, which shows that proactive message delivery contributes to performance. However, broadcasting to all agents performs even worse, with a mean score of 14.6404.

The communication statistics help distinguish message selectivity from message frequency. One-to-one send generates 90.41 send calls per task, whereas one-to-all send generates only 48.25. Their send-call rates per 100 model responses are similar, at 8.57 and 8.16, respectively. Thus, the poorer performance of one-to-all send cannot be explained simply by a higher normalized sending frequency. In the current architecture, directing messages to a selected recipient is more effective than broadcasting the same type of information to all agents.

Table 3: Communication ablations with GPT-5.6 Luna. (a) Query availability and scope on RepoZero-Py2JS Hard. (b) Send recipient scope on EdgeBench. Tokens, queries, and send calls are per-task means unless noted. M denotes millions. Bold indicates the highest scores and lowest token usage.

(a) Query availability and scope

Configuration	All-pass (%)	Micro (%)	Mean tokens (M)	Mean queries per task
Full query	**66.91**	**93.32**	3.53	13.23
Direct-family query	47.48	76.13	**2.56**	5.91
Query disabled	47.48	78.77	3.24	0.00

(b) Send recipients

Configuration	EdgeBench mean score	Mean send calls per task	Send calls per 100 responses
One-to-one send	**22.1010**	90.41	8.57
Send disabled	16.7868	0.00	0.00
One-to-all send	14.6404	48.25	8.16

5 Discussion

We analyze NanoMA's execution behavior from four perspectives. First, we measure the frequency, transitions, and target patterns of topology actions in execution traces. We then characterize the diversity of the orchestration structures formed by these actions. Next, we examine the task of RepoZero-Py2JS Hard to illustrate how autonomous delegation, communication, and parallel execution shape execution efficiency. Finally, we present a case study of autonomous parallel problem solving on a representative RepoZero-Py2JS Hard task.

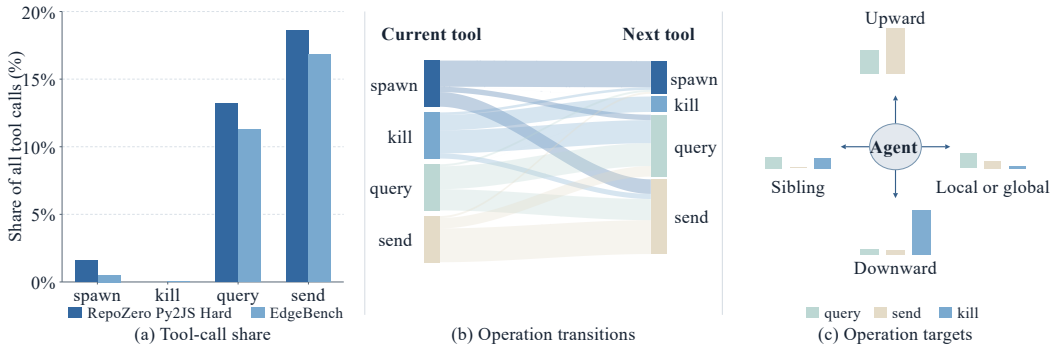


Figure 3: Topology actions observed in NanoMA trajectories. (a) Share of all tool calls by operation and benchmark. (b) First-order transitions between consecutive topology actions in the merged traces, normalized by the current action. (c) Target directions of `query`, `send`, and `kill` in EdgeBench, normalized within each topology action.

5.1 Topology Action Patterns

Figure 3(a) shows that information exchange dominates topology-action use. `send` accounts for 18.63% of all tool calls in RepoZero and 16.83% in EdgeBench, followed by `query`. In contrast, `spawn` accounts for only 1.58% and 0.58%, respectively, and `kill` is rare. Agents thus devote more actions to coordinating existing branches than to creating or terminating them. Panel (b) shows recurring sequences among consecutive topology actions. A `spawn` is followed by another `spawn` in 56.73% of cases, suggesting grouped delegation. Similarly, 73.18% of `send` actions are followed by another `send`, and 45.88% of `query` actions are followed by `send`. These patterns suggest that coordination often involves successive messages and communication following information retrieval. Panel (c) shows target patterns in EdgeBench. Upward messages constitute 75.6% of `send` calls, consistent with evidence aggregation by upstream agents. Queries reach multiple targets, including siblings and local or global state. Siblings receive 22.9% of queries, showing that information access extends beyond direct delegation links. Finally, 73.7% of successful `kill` operations target downstream agents, indicating that pruning mainly acts on delegated branches.

5.2 Orchestration Diversity

NanoMA's agents autonomously form orchestration structures according to task needs, rather than following predefined topology templates. Each agent decides when to delegate and which agents to query or inform as execution unfolds. In the analyzed RepoZero-Py2JS Hard and EdgeBench traces (Figure 4), the same twelve-category classification assigns NanoMA all twelve labels, including those assigned to Omnigent and CC-Workflow, which cover two and five, respectively. Ordinary stars account for 42.01% of NanoMA's records, alongside deeper delegation, branching trees, dense information-sharing patterns, and cyclic relations. This diversity arises from agents combining topology actions to meet evolving task requirements, not merely from adding workers around one coordinator. These descriptive findings come from unmatched cohorts and do not establish a performance ranking; formation mechanisms and classification criteria are given in Appendix B.

5.3 Efficiency Analysis

We further examine the relationship between task performance and resource usage on RepoZero-Py2JS Hard. Under GPT-5.6 Luna, NanoMA achieves the highest all-pass rate (66.91%) and Micro score (93.32%). Its token usage is 490.69M, which is lower than that of CC-Workflow (682.78M), and its execution time is substantially shorter. The other backbones show different trade-offs: under Claude Sonnet 5, NanoMA leads both metrics at greater token and time cost than some baselines, while under Qwen3.7-Plus, it achieves the highest Micro score with fewer tokens than Omnigent, but a lower all-pass rate. Taken together, these results show that NanoMA can achieve leading or near-leading task performance with competitive resource usage on most evaluated backbones, although

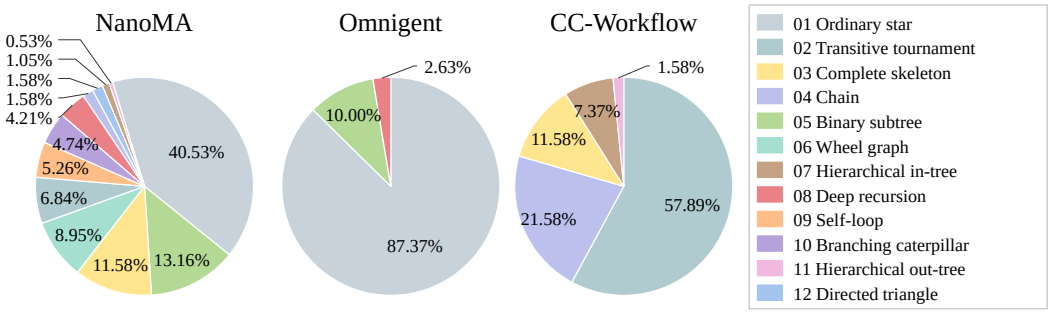


Figure 4: Primary-label distributions under the twelve-category rule. Each task record receives one label; categories include whole-graph patterns and local motifs. "Directed triangle" denotes a directed 3-cycle. Cohorts differ, and Omnigent includes RepoZero only. See Appendix B.

the performance efficiency relationship depends on the backbone and comparison framework. Detailed statistics are reported in Appendix C.

5.4 Case Study: Autonomous Parallel Problem Solving

The RepoZero-Py2JS Hard task `schedule/test16.py` illustrates how autonomous multi-agent orchestration supports parallel problem solving. The task requires porting Python logic for bulk creation, tag-based lookup, cancellation, and clearing of scheduled jobs to JavaScript while preserving the original outputs. Under GPT-5.6 Luna, all three frameworks pass all 156 tests. NanoMA completes the task in 161.3 seconds, compared with 1099.5 seconds for the central-planning baseline CC-Workflow and 582.9 seconds for the node-recursive baseline Omnigent. This corresponds to speedups of $6.82\times$ and $3.61\times$, respectively. NanoMA therefore reaches the same fully passing result in substantially less time. Its trace contains nine agents across two delegation levels and up to eight concurrent LLM requests. At least four requests overlap during 82.8% of the generation phase. Implementation, semantic analysis, and validation proceed concurrently, while child agents further decompose their assignments. This sustained overlap shows that parallel execution continues beyond the initial task decomposition.

The trace shows how topology actions support this execution pattern. The root creates implementation, semantic-analysis, and validation branches, after which the implementation branch decomposes its work on the `Job` object and job-collection management. Agents use `query` to retrieve progress and `send` to exchange evidence and corrections. During validation, one branch identifies a failure in the cancellation formula when more than ten jobs are cancelled, and another derives a correction. The root confirms that the implementation incorporates the correction. The final candidate is then reviewed, and the run ends at 161.3 seconds. This sequence illustrates recursive delegation, cross-branch information access, and parallel validation within the same successful execution.

6 Conclusion

Long-horizon tasks require multi-agent systems to revise their division of labor and collaboration as execution unfolds. NanoMA places two types of actions in every agent's ReAct loop: execution through `work`, and topology control through `spawn`, `kill`, `query`, and `send`. By decoupling task provenance from collaboration, the execution graph can expand, exchange information, and contract in response to runtime evidence. On RepoZero-Py2JS Hard, NanoMA achieves the highest all-pass and Micro scores with GPT-5.6 Luna High and Claude Sonnet 5, and retains the highest Micro score with Qwen3.7-Plus; on EdgeBench, it obtains the highest overall score for both evaluated models. Query ablations show that cross-branch information access contributes to task correctness and that targeted messages outperform broadcasting. Together, these results indicate that autonomous orchestration improves both the diversity of collaboration structures and task effectiveness, while the trade-off between efficiency and effectiveness remains a direction worth exploring.

Artificial Intelligence (AI) Use Statement

Generative AI tools assisted with literature search and synthesis, translation, manuscript drafting and editing, formalizing the method description, and interpreting experimental results. AI tools also assisted with code debugging and correction. The experimental design was developed entirely by the human authors. All AI-assisted content, including suggested code corrections, was reviewed by the authors. The authors take full responsibility for the final manuscript, its claims, and the associated research artifacts.

Ethics Statement

This work evaluates autonomous multi-agent systems on existing software-generation and long-horizon task benchmarks. Dynamic delegation and tool execution can amplify unintended actions, resource consumption, and exposure of sensitive information. The runtime controls described in Appendix A address execution management and resource limits, but do not constitute comprehensive security guarantees. Deployment beyond benchmark environments should therefore restrict tool permissions, protect credentials and sensitive data, and retain human oversight for consequential actions. The reported performance gains should not be interpreted as evidence of safe unrestricted autonomy.

Reproducibility Statement

The Experiments section describes the benchmarks, evaluation metrics, and framework comparisons. Appendix A documents the runtime, tool semantics, communication mechanisms, and agent lifecycle controls. Sections 4.3.1 and 4.3.2 describe the communication ablations, while Appendix C reports resource consumption on RepoZero-Py2JS Hard. An anonymous implementation repository is linked in the abstract. These materials support inspection of the method and its evaluation; exact outcomes may vary with model-service updates and nondeterministic execution.

References

Anthropic. Orchestrate teams of Claude Code sessions. Claude Code Documentation, 2026. URL https://code.claude.com/docs/en/agent-teams. Accessed September 20, 2026.

OpenAI. Subagents. ChatGPT Learn Documentation, 2026. URL https://learn.chatgpt.com/docs/agent-configuration/subagents. Accessed September 20, 2026.

Deyao Zhu, Xin Zhou, Shengling Qin, Xuekai Zhu, Hangliang Ding, Shu Zhong, Zixin Wen, Zhonglin Xie, Chenhui Gou, Linxuan Ren, Yueyang Wang, Junfeng Zhong, Rui Liu, Tian Gao, Yangguang Lin, Jingyuan Zhang, Maojia Song, Xuan Qi, Jinhong Wu, Chenyang Zhang, Yinzhu Piao, Ziru Niu, Hongbin Lin, Lingxiang Meng, Peng Tang, Chengyao Tang, Shanyu Wu, Huanyu Zheng, Yu Liu, Liya Zhu, He Wang, Ming Ding, Ziyu Wan, Hao Liu, Sibo Wang, Haotian Zhu, Xintian Zhang, Nan Chai, Yipeng Liu, Panhao Lai, Sihang Yuan, Zixin Su, Ge Zhang, Wangchunshu Zhou, Yantao Du, Wenhao Huang, and Guang Shi. EdgeBench: Unveiling scaling laws of learning from real-world environments. *arXiv preprint arXiv:2607.05155*, 2026. doi: 10.48550/arXiv.2607.05155. URL https://arxiv.org/abs/2607.05155.

Adam Fourney, Gagan Bansal, Hussein Mozannar, Cheng Tan, Eduardo Salinas, Erkang Zhu, Friederike Niedtner, Grace Proebsting, Griffin Bassman, Jack Gerrits, Jacob Alber, Peter Chang, Ricky Loynd, Robert West, Victor Dibia, Ahmed Awadallah, Ece Kamar, Rafah Hosn, and Saleema Amershi. Magentic-One: A generalist multi-agent system for solving complex tasks. *arXiv preprint arXiv:2411.04468*, 2024. doi: 10.48550/arXiv.2411.04468. URL https://arxiv.org/abs/2411.04468.

Yufan Dang, Chen Qian, Xueheng Luo, Jingru Fan, Zihao Xie, Ruijie Shi, Weize Chen, Cheng Yang, Xiaoyin Che, Ye Tian, Xuantang Xiong, Lei Han, Zhiyuan Liu, and Maosong Sun. Multi-agent collaboration via evolving orchestration. In *Advances in Neural Information Processing Systems*, volume 38, pp. 165025–165059. Curran Associates, Inc., 2025. doi: 10.52202/085713-5502. URL https://proceedings.neurips.cc/paper_files/paper/2025/file/f1320d2e2842169c6fc89dcbd80e94d0-Paper-Conference.pdf.

Jianhao Ruan, Zhihao Xu, Yiran Peng, Fashen Ren, Zhaoyang Yu, Xinbing Liang, Jinyu Xiang, Yongru Chen, Bang Liu, Chenglin Wu, Yuyu Luo, and Jiayi Zhang. AOrchestra: Automating sub-agent creation for agentic orchestration. *arXiv preprint arXiv:2602.03786*, 2026. doi: 10.48550/arXiv.2602.03786. URL https://arxiv.org/abs/2602.03786v2. Version 2, revised February 7, 2026.

Yuzhen Mao and Azalia Mirhoseini. Decentralized multi-agent systems with shared context. *arXiv preprint arXiv:2606.10662*, 2026. doi: 10.48550/arXiv.2606.10662. URL https://arxiv.org/abs/2606.10662.

Andrew Zhu, Liam Dugan, and Chris Callison-Burch. ReDel: A toolkit for LLM-powered recursive multi-agent systems. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pp. 162–171, Miami, Florida, USA, 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.emnlp-demo.17. URL https://aclanthology.org/2024.emnlp-demo.17/.

Apurva Gandhi, Satyaki Chakraborty, Xiangjun Wang, Aviral Kumar, and Graham Neubig. Recursive agent optimization. *arXiv preprint arXiv:2605.06639*, 2026. doi: 10.48550/arXiv.2605.06639. URL https://arxiv.org/abs/2605.06639.

Salaheddin Alzu'bi, Baran Nama, Arda Kaz, Anushri Eswaran, Weiyuan Chen, Sarvesh Khetan, Rishab Bala, Tu Vu, and Sewoong Oh. ROMA: Recursive open meta-agent framework for long-horizon multi-agent systems. *arXiv preprint arXiv:2602.01848*, 2026. doi: 10.48550/arXiv.2602.01848. URL https://arxiv.org/abs/2602.01848v2. Version 2, revised February 14, 2026.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. ReAct: Synergizing reasoning and acting in language models. In *The Eleventh International Conference on Learning Representations*, 2023. URL https://arxiv.org/abs/2210.03629.

Zhaoxi Zhang, Yiming Xu, Jiahui Liang, Weikang Li, Xiaoshuai Chen, Liwei Qian, Xin Pei, Jizhou Huang, Run Sun, and Yunfang Wu. RepoZero: Can LLMs generate a code repository from scratch? *arXiv preprint arXiv:2605.07122*, 2026. doi: 10.48550/arXiv.2605.07122. URL https://arxiv.org/abs/2605.07122v3. Version 3, revised May 20, 2026.

Wentao Zhang, Ce Cui, Yilei Zhao, Yang Liu, and Bo An. Agentorchestra: A hierarchical multi-agent framework for general-purpose task solving. *arXiv preprint arXiv:2506.12508*, 2025. URL https://arxiv.org/abs/2506.12508.

Michael Rizvi-Martel, Satwik Bhattamishra, Neil Rathi, Guillaume Rabusseau, and Michael Hahn. Benefits and limitations of communication in multi-agent reasoning. *arXiv preprint arXiv:2510.13903*, 2025. URL https://arxiv.org/abs/2510.13903v2. Version 2, revised July 14, 2026.

Elias Lumer, Sahil Sen, Kevin Paul, and Vamse Kumar Subbiah. Recursive agent harnesses. *arXiv preprint arXiv:2606.13643*, 2026. doi: 10.48550/arXiv.2606.13643. URL https://arxiv.org/abs/2606.13643.

Omnigent. Omnigent agent YAML specification. Online documentation, 2026. URL https://github.com/omnigent-ai/omnigent/blob/main/docs/AGENT_YAML_SPEC.md. Accessed September 20, 2026.

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, Cambridge, MA, 4th edition, 2022. ISBN 978-0-262-04630-5. URL https://mitpress.mit.edu/9780262046305/introduction-to-algorithms/.

OpenAI. GPT-5.6 luna model. OpenAI API Documentation, 2026. URL https://developers.openai.com/api/docs/models/gpt-5.6-luna. Accessed September 2, 2026.

Anthropic. Introducing claude sonnet 5. Anthropic Product Announcement, 2026. URL https://www.anthropic.com/news/claude-sonnet-5. Accessed September 2, 2026.

Alibaba Cloud. Qwen3.7-Plus model information. Alibaba Cloud Model Studio Documentation, 2026. URL https://help.aliyun.com/en/model-studio/qwen3-7-plus. Accessed September 14, 2026.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, Hoang H. Tran, Fuqiang Li, Ren Ma, Mingzhang Zheng, Bill Qian, Yanjun Shao, Niklas Muennighoff, Yizhe Zhang, Binyuan Hui, Junyang Lin, Robert Brennan, Hao Peng, Heng Ji, and Graham Neubig. OpenHands: An open platform for AI software developers as generalist agents. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=OJd3ayDDoF.

OpenAI. Code generation with Codex. OpenAI API Documentation, 2026. URL https://developers.openai.com/api/docs/guides/code-generation. Accessed September 14, 2026.

CrewAI. CrewAI documentation. Online documentation, 2026. URL https://docs.crewai.com/. Accessed September 2, 2026.

Anthropic. Introducing dynamic workflows in claude code. Anthropic Product Announcement, 2026. URL https://claude.com/blog/introducing-dynamic-workflows-in-claude-code. Accessed August 28, 2026.

Illya Koval and Matthew Kwan. Exponentially many graphs are determined by their spectrum. *The Quarterly Journal of Mathematics*, 2024. URL https://arxiv.org/abs/2309.09788.

Robert Sedgewick and Kevin Wayne. *Algorithms*. Addison-Wesley, 4 edition, 2011. URL https://algs4.cs.princeton.edu/42digraph/.

David M. Mount. CMSC 420 Data Structures: Rooted trees and binary trees. University of Maryland course notes, Lecture 3, 2019. URL https://www.cs.umd.edu/class/fall2019/cmsc420-0201/Lects/cmsc420-fall2019-lects.pdf.

Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM*, 46(5):720–748, 1999.

Frank Harary and Allen J. Schwenk. The number of caterpillars. *Discrete Mathematics*, 6(4):359–365, 1973. doi: 10.1016/0012-365X(73)90067-8.

John W. Moon. *Topics on Tournaments*. Holt, Rinehart and Winston, 1968.

Coen Bron and Joep Kerbosch. Algorithm 457: Finding all cliques of an undirected graph. *Communications of the ACM*, 16(9):575–577, 1973. doi: 10.1145/362342.362367.

Shunya Tamura and Yuuho Tanaka. Number of spanning trees in a wheel graph with two identified vertices via hitting times. *arXiv preprint arXiv:2502.01965*, 2025. URL https://arxiv.org/abs/2502.01965.

Jack Edmonds. Optimum branchings. *Journal of Research of the National Bureau of Standards, Section B: Mathematics and Mathematical Physics*, 71B(4):233–240, 1967. doi: 10.6028/jres.071B.032. URL https://doi.org/10.6028/jres.071B.032.

A TECHNICAL DETAILS OF NANOMA

A.1 RUNTIME STATE, DYNAMIC EXECUTION GRAPH, AND NODE-LOCAL PLANNING

The graph and state notation from Section 3.1 is

$$G_t = (V_t, E_t^{\text{prov}}, E_t^{\text{comm}}), \tag{1}$$

with immutable creation links and dynamically established communication links. Each agent maintains

$$s_i^t = \left(\tau_i^t, h_i^t, Q_i^t, W_i^t, \ell_i^t, b_i^t, O_i^t, I_i^t\right), \tag{2}$$

whose components are defined in Section 3.4. The runtime additionally maintains a global agent registry, a durable candidate ledger, and shared resource state; I_i^t is implemented as priority-ordered inboxes.

The local task list stores `pending`, `in_progress`, `completed`, and `cancelled` entries; Q_i^t contains only unresolved entries. Before each turn, the active plan is injected as temporary context rather than appended repeatedly to history. A child starts with its own context, private workspace, and empty task list, not a copy of its parent's internal plan. The parent supplies the assignment, while the child controls its subsequent planning. Cross-branch access depends on the communication configuration described in Section A.4. An empty Q_i^t may indicate an unplanned task or no remaining listed work; normal completion additionally requires $\text{Valid} = 1$. Conversely, a queue with no `pending` entries may still contain resumable `in_progress` work.

A.2 TOPOLOGY ACTIONS AND NODE-LEVEL SPAWNING

The four topology actions introduced in the main text correspond to the implementation interfaces summarized in Table 4.

Table 4: NanoMA's four topology actions, their data-structure roles, and implementation adapters.

Data-structure role	Topology action	Engineering adapters
Insertion	`spawn`	SPAWNJUDGE and configuration-specific batched or task-bound insertion interfaces
Deletion	`kill`	Automatic pruning policies
Lookup	`query`	`wait` and status/resource inspection
Update	`send`	`deliver_to_parent` and artifact-transfer interfaces

`spawn` inserts a direct child for an explicit subtask. In the same-model judge profile, `task_create` proposes work to SPAWNJUDGE at a fresh planning point; model approval and runtime admission authorize creation. Otherwise, the item remains local. Workers have no unrestricted direct-spawn schema; direct, batched, and task-bound adapters in other profiles invoke the same primitive. Children inherit action capabilities subject to policy. `delegate` additionally ends the parent and transfers responsibility.

Planning prompts request a bounded objective, expected output, and relation to existing work. They guide local decomposition without guaranteeing its utility or specifying a global plan. Runtime checks enforce assignment and resource validity. Figure 5(a) summarizes this loop.

The action set available to agent i at time t is

$$\mathcal{A}_i^t = \text{Guard}\big(\text{Policy}\left(\mathcal{A}_0, s_i^t, S_t, C\right)\big), \tag{3}$$

where $\mathcal{A}_0$ is the registry, S_t the global runtime state, and C the configuration. Policy responds to resources, dependencies, missing outputs, and stagnation; guards enforce hard lifecycle and expansion limits. These checks constrain available actions without specifying a global plan.

Algorithm 1 summarizes the agent's decision cases, not a mandatory sequence of tool calls. Completion is checked before task selection, which requires a nonempty ready set. SPAWNJUDGE supplies the model-side decomposition decision, whereas $\text{admit}_{\text{spawn}}(i, t)$ records the separate runtime admission decision. Runtime stop and failure conditions in Table 6 can interrupt any branch.

Algorithm 1: Node-local execution and topology actions in NanoMA's multi-agent ReAct loop.

Input: active node v_i, local state s_i^t, runtime state S_t, and configuration C.
Output: updated local plan Q_i, execution graph G_t, and node lifecycle state ℓ_i.

while $\ell_i =$ `running` **do**
  Incorporate tool results and messages; inspect visible agents and refresh Q_i as needed.
  if $Q_i = \varnothing$ and $\mathrm{Valid}(\tau_i, h_i, W_i, O_i) = 1$ **then**
    Deliver under the output contract; request guarded `done`; **continue**.
  Let $R_i = \mathrm{Ready}(Q_i, I_i)$.
  if $R_i \neq \varnothing$ **then** select $u \in R_i$, including resumable `in_progress` work.
    if a fresh decomposition is proposed, SPAWNJUDGE(u, s_i^t), and $\mathrm{admit}_{\mathrm{spawn}}(i, t)$ **then**
      Create child $v_j \leftarrow \mathrm{Runtime.spawn}(u)$ and record the delegation.
    else perform local `work` on u and update its task state.
  else if $Q_i = \varnothing$ **then** plan, create tasks, or perform direct `work` without selecting u.
  else reconcile dependencies through communication or `wait`; request `idle` if the yielding criterion holds.
  if $\ell_i =$ `running` **then** use `query`, `send`, or `kill` as needed; revise Q_i from feedback.
end while

A.3 Complete Tool Surface and Functional Roles

Table 5 groups the 28 model-facing tools in the current worker registry before policy filtering. Functional families are not additional ReAct action types. Benchmark adapters may add environment-specific tools; the internal `spawn` primitive is described separately in Section A.2.

For the direct effects referenced in Section 3.3, Table 6 maps successful calls to changes in s_i^t and G_t. In particular, `query` updates only the caller's observation, history, and budget; `send` adds a recipient message and a communication edge, with context injection governed by delivery timing; `spawn` creates an agent and provenance edge; and `kill` ends an active agent without deleting its historical graph records.

Table 5: Grouped model-facing tool registry in the current NanoMA profile. Families describe implementation functions, not additional ReAct action types.

Family	Interface	Primary role and state effect
Planning	`task_create`, `task_update`, `task_list`	Create, revise, and inspect local tasks; creation may trigger SPAWNJUDGE.
Coordination	`query`	Discover visible agents and read state, results, artifacts, and bounded history.
Coordination	`send`	Deliver an addressed message and wake an idle recipient.
Coordination	`wait`	Wait for all or any selected children; return completed and pending agents without assuming success.
Coordination	`deliver_to_parent`	Record a child's answer and evidence durably, then notify its parent.
Coordination	`transfer`, `deliveries`	Transfer artifacts or inspect and collect pending child changes through validation.
Lifecycle	`set_bio`, `get_cost`	Publish a role description and inspect resource state.
Lifecycle	`set_status`	Request `idle` or `done`, subject to enabled completion guards.
Lifecycle	`kill`	End the caller or a descendant after ancestry and delivery-grace checks.
Lifecycle	`rebirth`	Reset context from a summary while preserving agent identity, workspace, and tool access.
Execution	`shell`, `batch`	Execute commands or dispatch registered calls subject to policy checks.

Continued on the next page

Table 5 (continued)

Family	Interface	Primary role and state effect
File input/output	`ws_create_file`, `ws_append_file`, `ws_read_file`	Create, append, and read workspace files.
File editing	`ws_replace_string`, `ws_multi_replace`, `ws_apply_patch`	Apply guarded replacements and structured patches.
Code search	`ws_grep`, `ws_code_outline`, `ws_read_symbol`	Search text, inspect structure, and retrieve symbols.
Validation	`verify`, `experiments`	Register reproducible checks, inspect measurements, and restore retained states.
Publication	`submit`	Publish artifacts through the configured root-owned delivery contract.

`deliver_to_parent`, `transfer`, and `deliveries` are typed implementations of information or artifact delivery within the `send` abstraction, not extra topology actions. Their persistence and validation semantics are summarized in Section A.6.

A.4 Node Discovery, Communication, and Synchronization

`query` reads the registry without changing its targets. A listing returns visible identifiers, lifecycle states, and descriptions; a targeted query can additionally return relations, results, artifacts, and bounded history. The ablation in Section 4.3.1 compares full cross-branch access, direct-family access to the caller, its parent, and its direct children, and disabled query. These are information-access policies, not changes to the meaning of a successful read.

The standard `send` interface accepts explicit recipient identifiers. Section 4.3.2 contrasts selected-recipient communication with a one-to-all configuration that broadcasts to other agents without recipient selection, and with disabled send. Recipient selection is distinct from delivery timing. A message is represented as $m = (i, j, x, \rho, t)$, where i and j are the sender and receiver, x is the content, and ρ is the delivery priority. The current implementation supports three priorities:

- `queue`: injected before the receiver's next turn;
- `steer`: injected after the current group of tool calls and before the next model invocation; and
- `immediate`: interrupts tool calls that have not yet executed and injects the message immediately.

Sending a message to an `idle` node wakes it. The `wait` operation provides explicit synchronization and may wait for either all targets or the first target to complete. Waiting terminates when a relevant message or candidate arrives, a requested completion condition is met, or the timeout expires. Nodes may therefore implement conventional parent–child synchronization while incrementally consuming asynchronous results from different branches.

The reported ablations concern query availability and send recipients; they do not isolate the effects of the three message priorities.

A.5 Bounded Expansion and Lifecycle Control

Implemented states are `running`, `idle`, `done`, and `failed`; the diagram's `new` and `pruned` mark events, with accepted pruning mapping to `done`. Normal completion permits `running`→`done`; pruning or a runtime stop also permits `idle`→`done` without resuming ReAct. Thus, `done` records termination, not necessarily success. In Section 3.4, Ready denotes executable work and Valid an evidence-based completion assessment, not a correctness oracle or exhaustive runtime guards. Table 6 covers guarded requests, external stops, and exceptions using the state and graph variables from Sections 3.4 and 3.1. Its final column lists only potentially changed terms; v_j is an affected child, recipient, or pruning target.

Table 6: Principal implemented node and graph transitions in NanoMA. Applicable completion guards are evaluated before a requested terminal transition is committed.

Current ℓ	Implemented condition	Event or operation	Next ℓ	Observable state or graph update
Not registered	The runtime creates the root node and starts its execution task.	Register and start	`running`	Add v_i to V_t and initialize $\tau_i^t, h_i^t, Q_i^t, W_i^t, \ell_i^t, b_i^t, O_i^t, I_i^t$.
`running`; $v_j \notin V_t$	`spawn` receives a non-empty task and passes the depth, node-count, memory, model, and policy guards.	`spawn`	i: `running`; j: `running`	Add v_j to V_t, add (v_i, v_j) to E_t^{prov}, initialize s_j^{t+1}, and update h_i^{t+1} and b_i^{t+1}.
`running`	A normal execution or planning tool returns and no terminal condition fires.	Work or plan update	`running`	Update h_i^{t+1} and b_i^{t+1}; a successful action may also update Q_i^{t+1}, W_i^{t+1}, or O_i^{t+1}.
`running`	`rebirth` receives a non-empty summary and is applied at the start of the next turn.	Context reset	`running`	Archive the old h_i^t in W_i, replace it with the system context and summary, optionally update τ_i^t, and reset the context component of b_i; node identity, Q_i, workspace, lifecycle, and prior results are preserved.
`running`	A registry listing is requested, or the requested target exists.	`query`	`running`	Update the caller's o_i^t, h_i^{t+1}, and b_i^{t+1}; the target state and G_t are unchanged.
`running`	The message is non-empty, its mode is `queue`, `steer`, or `immediate`, and the recipient exists.	`send`	Sender remains `running`	Add the envelope to I_j^{t+1}, realize (v_i, v_j) in E_{t+1}^{comm}, and update h_i^{t+1} and b_i^{t+1}.
`running`	The selected `all`/`any` condition is met, a message or candidate arrives, or the bounded timeout expires.	`wait`	`running`	Return completed and pending nodes through o_i^t, then update h_i^{t+1} and b_i^{t+1}; a concurrent message may also update I_i^{t+1}.
`running`	`set_status(idle)` is accepted; a forced finalization phase may reject this request.	Yield	`idle`	Update ℓ_i^{t+1}, h_i^{t+1}, and b_i^{t+1}; the node keeps its task, history, workspace, and results, while a non-empty result argument may update O_i^{t+1}.
`idle`	An addressed `queue`, `steer`, or `immediate` envelope is delivered.	Message wake-up	`running`	Update I_i^{t+1} and ℓ_i^{t+1}, restart the execution task if needed, and copy the message into h_i when it is injected.
`running`	An enabled guard rejects completion because fixed-profile child work or evidence is unresolved, the root delivery contract is incomplete, or a first merge-overlap check finds live peers.	Reject completion	`running`	Add the rejection and required next action to h_i^{t+1}, update b_i^{t+1}, and leave ℓ_i and G_t unchanged.
`running`	`set_status(done)` passes all enabled guards, or `submit` succeeds while forced finalization is active. Ordinary `submit` alone does not terminate a node.	Finish	`done`	Update ℓ_i^{t+1}, O_i^{t+1}, h_i^{t+1}, and b_i^{t+1}; a child completion also sends a terminal envelope to its parent's inbox.
`running`	A child with a live parent supplies a non-empty answer that passes any configured answer-shape check.	Deliver	`done` if auto-completion is enabled; otherwise `running`	Update O_i^{t+1}, the candidate ledger, I_j^{t+1}, and E_{t+1}^{comm}; update ℓ_i^{t+1} only when auto-completion is enabled.
`running`; $v_j \notin V_t$	An admitted `spawn` sets `delegate=true`.	Spawn and delegate	i: `done`; j: `running`	Apply the spawn updates above, then update ℓ_i^{t+1} and O_i^{t+1} with the delegated child identifier.
`running` or `idle`	A descendant has material evidence but no formal delivery, and this is the first kill request or no additional target turn has completed.	Delivery-grace request	Unchanged	Add a `steer` envelope to I_j^{t+1}, realize the communication edge, and update the caller's h_i^{t+1} and b_i^{t+1}; the active frontier is unchanged.
`running` or `idle`	The caller targets itself or one selected descendant, and delivery grace is inapplicable or has completed before a forced retry.	`kill`	`done`	Update only the selected target's ℓ_j^{t+1}, remove it from the active frontier, and cancel its execution task when needed. V_t, existing edges, and the target's descendants remain unchanged.

Continued on the next page

Table 6 (continued)

Current ℓ	Implemented condition	Event or operation	Next ℓ	Observable state or graph update
`running` or `idle`	A per-node turn or time limit, the global total-token limit, a configured role deadline, probe stop, or runtime shutdown is reached.	Runtime stop	`done`	Update ℓ_i^{t+1}, b_i^{t+1}, and any fallback O_i^{t+1}; final delivery and parent notification run during cleanup when applicable.
`running`	The global budget is exhausted, malformed tool output exceeds its repair window, or an execution exception remains unrecovered.	Runtime failure	`failed`	Update ℓ_i^{t+1}, h_i^{t+1}, b_i^{t+1}, and diagnostic O_i^{t+1}; notify the parent through its inbox when one exists.
`done` or `failed`	Ordinary execution after a terminal event.	Absorb	Unchanged	The lifecycle state is unchanged. A later envelope may be queued in I_i, but it does not restart the execution task.

ReAct runs only in `running`; `idle` suspends model calls until an addressed message restores `running`. A bounded `wait` call returns to `running` without changing the lifecycle state. Rejected completion also leaves the agent `running`, with feedback for replanning. Explicit snapshot restoration is outside this transition relation.

A.5.1 Operational Local-Liveness Window

Progress changes plans, workspace, results, collaboration, or lifecycle state; repeated reasoning, queries, or wait timeouts without such changes do not qualify. The runtime tracks inactivity, resources, dependencies, missing outputs, and progress signatures. Configured thresholds can bias or restrict actions toward execution, communication, synchronization, restructuring, or termination. This stagnation policy neither imposes a universal timeout nor guarantees useful model progress.

A `spawn` operation is admitted only when all of the following conditions hold:

$$\begin{aligned}\mathrm{admit}_{\mathrm{spawn}} = \mathbf{1}[d_i + 1 \le D_{\max}]\mathbf{1}[|V_t| < N_{\max}] \\ \cdot \mathbf{1}[n_t^{\mathrm{active}} < C_{\mathrm{mem}}]\mathbf{1}[m \in \mathcal{M}_{\mathrm{allow}}]\mathbf{1}[\text{policy permits}],\end{aligned} \tag{4}$$

where $D_{\max}$, $N_{\max}$, C_{mem}, and $\mathcal{M}_{\mathrm{allow}}$ are the depth limit, agent limit, memory-derived active capacity, and allowed model set. Agent count is distinct from model-request concurrency, which is limited separately by C_{LLM}. Token, wall-clock, turn, and monetary budgets provide additional independent bounds.

A.5.2 Pruning Semantics Under Concurrent Activity

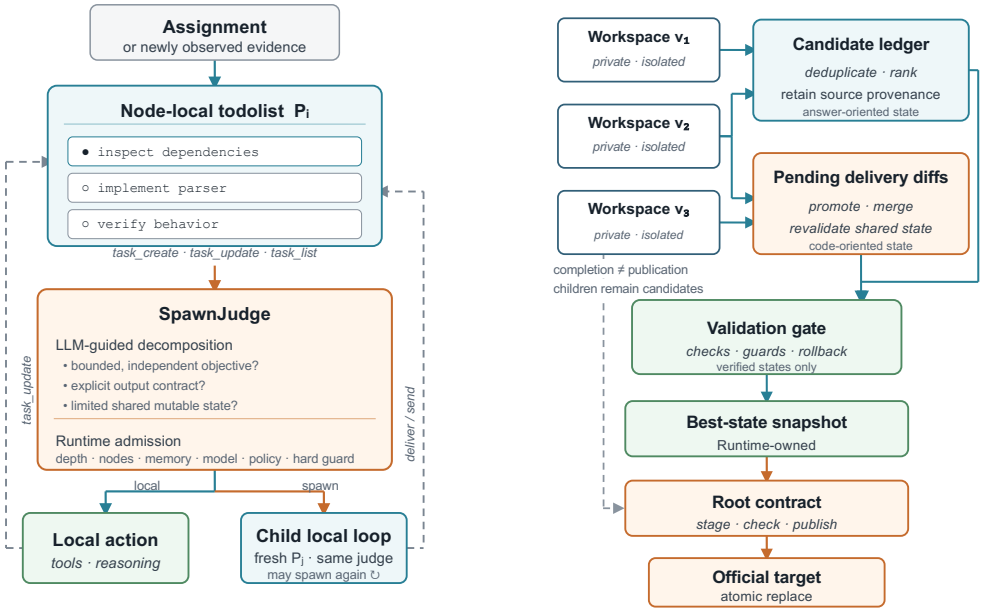
`kill` may target only the caller or a descendant. If the target has material evidence but no formal delivery, the runtime requests delivery and defers termination until an additional execution opportunity has been given. This grace period is intended to preserve evidence before pruning.

Define the active frontier as $\mathcal{F}_t = \{v_i \in V_t : \ell_i^t \in \{\texttt{running}, \texttt{idle}\}\}$. An accepted `kill` sets active agent v_j to `done`, giving $\mathcal{F}' = \mathcal{F} \setminus \{v_j\}$. "Graph contraction" thus refers to the active execution structure, not deletion from the historical graph G_t: the agent and its recorded creation and communication edges remain. Descendants are not recursively terminated. Concurrent spawning can offset this local decrease, so convergence concerns ordered committed transitions rather than a monotonically decreasing agent count. Waking an idle agent leaves frontier membership unchanged.

In fixed-orchestration profiles, completion guards can require unresolved children to be inspected, awaited, redirected, or pruned before the parent finishes, subject to configured evidence and deadline exceptions.

A.5.3 Scope of the Structural-Convergence Claim

Convergence in Section 3.4 assumes a strict well-founded order $\prec$ over unfinished obligations: each agent creates finitely many children with smaller obligations; each progress transition terminates or strictly reduces its obligation; and stalled agents eventually resume useful work or terminate. Idling alone is insufficient, and terminal agents do not reactivate during ordinary execution. These decomposition and fairness assumptions exclude infinite spawning, descent, and stagnation. They



(a) Node-local planning and expansion **(b)** Candidate delivery and publication

Figure 5: Planning and delivery in NanoMA. (a) Each agent uses its local task list and SPAWNJUDGE to choose `work` or create a child with the same planning capability. (b) Answer and code-state paths converge through validation and root-owned publication.

imply finitely many agents, each eventually terminating, so some finite T satisfies $\mathcal{F}_T = \varnothing$. This establishes structural termination under the stated conditions, not answer correctness.

The argument requires neither a fixed topology nor hard depth or agent caps. Those caps, resource limits, malformed-output repair, exception handling, and shutdown cleanup provide protection when budgets are strict or assumptions fail. Limit-triggered termination alone is not endogenous convergence.

A.6 CANDIDATE DELIVERY, WORKSPACE ISOLATION, AND FINAL PUBLICATION

Answer candidates, code state, and final artifacts follow distinct delivery stages (Figure 5(b)).

Durable result delivery. A child agent uses `deliver_to_parent` to store its answer, evidence, confidence, and method in a durable ledger before notifying its parent through `steer`. Delayed message processing does not erase candidates; duplicates are consolidated, and weaker recovered records cannot replace formal deliveries. Configured delivery can also complete the child.

Workspace isolation and merging. In merge-delivery configurations, children receive private executable task trees; only submission paths enter baseline-relative diffs. Merging uses a lock and snapshot and may finish after child completion. Overlapping edits are reported, not automatically three-way merged; later edits can overwrite earlier ones. Registered checks validate merged states, restoring the snapshot on explicit failure and retaining verified non-regressing states independently of agent context. Without a usable check, a state cannot become the verified best.

Final publication. The root's delivery contract specifies required files, an optional validator, and the destination. Publication checks completeness and configured validation before atomic replace-

ment, with failure recovery. Children cannot bypass root reconciliation; recording, merging, and publication are not one transaction.

B Orchestration Structures: Formation and Classification

Figure 6 illustrates the twelve categories defined below.

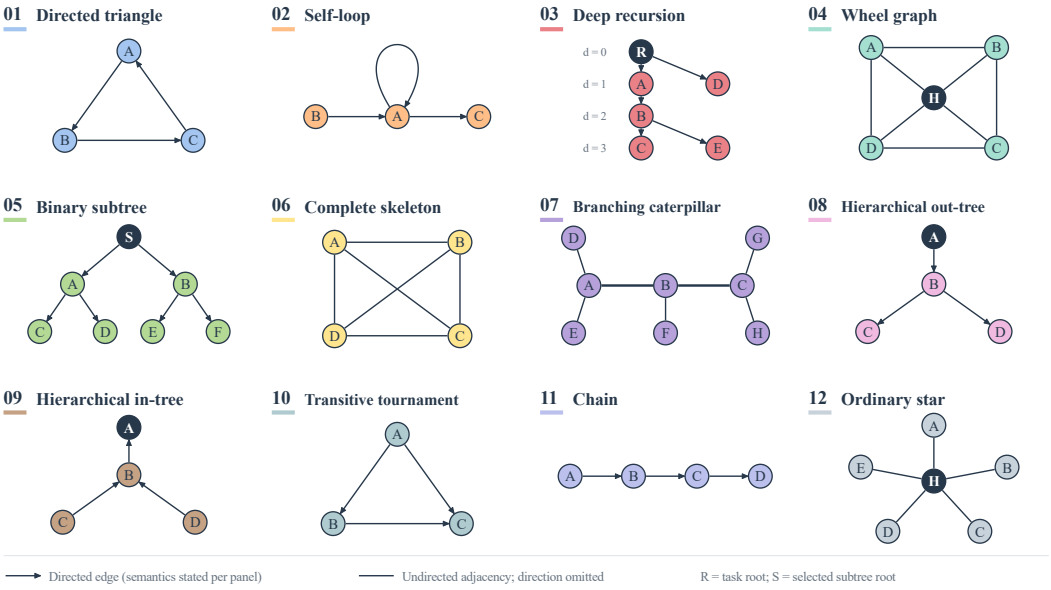


Figure 6: Twelve descriptive orchestration structures, not predefined templates. Arrows indicate directed relations and lines undirected adjacency; relation layers and classification criteria are defined below.

B.1 Graph Representation and Theoretical Basis

For execution r, C_r, D_r, and K_r record creation, source-to-consumer information flow, and enforced execution precedence; $U(\cdot)$ removes directions and self-loops. Send edges point to recipients; query-result edges point from targets to callers. Unlike E_t^{comm} in Section 3.1, D_r includes observed reads and result dependencies. Only recorded completion dependencies such as `await` establish K_r, not information flow alone. Detection rules are analysis-specific, not exhaustive, and do not establish agent intent or performance benefits.

B.2 Delegation and Control Structures

Ordinary star. A star $K_{1,k}$ connects a center to mutually unconnected leaves (Koval & Kwan, 2024), through child creation or information exchange. The test accepts a whole creation star with at least three agents or an induced three-agent information in-/out-star, without implying broadcast or reciprocity.

Chain. A chain passes results along an induced three-agent, two-edge directed path (Sedgewick & Wayne, 2011). Parallel work remains possible; sequential control additionally requires enforced predecessor completion.

Binary subtree. Local two-way decomposition can form a binary tree (Mount, 2019). The test retains an agent and *all* descendants in C_r, requiring an arborescence with maximum out-degree exactly two. A single two-child split qualifies; balance, completeness, and additional depth are unnecessary.

Deep recursion. Successive delegation produces recursive spawn paths (Blumofe & Leiserson, 1999). Counting the root at depth zero (Mount, 2019), the test requires an acyclic creation graph with a path of depth at least three—a classification convention, not recursion's theoretical boundary.

Branching caterpillar. A main delegation or information route with side contributors can form a caterpillar: a tree whose non-leaf vertices form a path after simultaneous leaf removal (Harary & Schwenk, 1973). The whole $U(C_r)$ or $U(D_r)$ must satisfy this condition, with maximum degree from three to $n-2$ to exclude paths and stars.

B.3 Information-Sharing Structures

Transitive tournament. The induced pattern $a \to b, b \to c, a \to c$ (Moon, 1968) forms when an agent receives original evidence and an intermediate contribution through queries, messages, or result use. The direct $a \to c$ edge distinguishes it from a chain; reciprocity is absent.

Complete skeleton. Pairwise information access can form a complete skeleton, or clique (Bron & Kerbosch, 1973). The whole-graph test requires $U(D_r) = K_n$, $n \geq 4$; the local test accepts a K_4 witness. Neither requires reciprocal edges. A local clique establishes dense group collaboration, not all-to-all connectivity across the entire execution.

Wheel graph. A wheel combines a shared hub with a peer-connected rim (Tamura & Tanaka, 2025), allowing integrator access and lateral exchange. The test searches $U(D_r)$ for a chordless rim of length four to eight and a hub adjacent to every rim vertex. Undirected rim connectivity does not establish a directed feedback cycle.

Hierarchical in-tree. An in-tree directs edges toward a root (Mount, 2019), as intermediate agents aggregate and forward evidence across levels. The test reverses the four-agent out-tree patterns below, requiring induced merging with height at least two. This is an information hierarchy and may cross provenance branches.

Hierarchical out-tree. An out-tree propagates information through successive recipients (Mount, 2019; Edmonds, 1967). Its induced patterns are $a \to b, a \to c, b \to d$ and $a \to b, b \to c, b \to d$, with no additional edges among these agents. Both branch across levels, but neither implies recursive agent creation.

Self-loop. A self-addressed message creates $v \to v$ (Sedgewick & Wayne, 2011). The test requires this explicit relation, not merely an agent's ReAct loop, and does not infer reflection, successful correction, or productive iteration from it.

Directed 3-cycle. Information relations $a \to b, b \to c, c \to a$ close a directed cycle (Sedgewick & Wayne, 2011). The induced test requires exactly these three edges, giving each agent in-degree and out-degree one. Episode-level aggregation establishes the cyclic relation, not necessarily a temporally ordered causal feedback loop.

B.4 Classification Logic and Compositional Interpretation

Whole-graph tests cover one episode or workflow; local tests select agents within it. Induced tests retain all edges among selected agents, and subtree tests retain all descendants. Neither arbitrary edge removal nor cross-episode aggregation is allowed.

Overlapping structures receive one primary label by first match, in this order: directed 3-cycle; self-loop; deep recursion; wheel; binary subtree; whole complete skeleton; branching caterpillar; hierarchical out-tree; hierarchical in-tree; transitive tournament; local complete skeleton; chain; ordinary star. Whole and local complete-skeleton tests share a label. This convention makes labels exclusive, not structures; search bounds do not limit harness expressivity.

`kill` changes activity without erasing classified relations. Recorded structures are therefore neither controller templates nor proof of simultaneous activity or causal benefit.

C RESOURCE COST ON REPOZERO-PY2JS HARD

Table 7 reports token consumption and agent counts for the three model backbones on all 139 RepoZero-Py2JS Hard tasks. Token totals and per-task averages are reported in millions. Table 8 complements these measurements with paired execution times for GPT-5.6 Luna and Claude Sonnet 5.

Table 7: Resource use on RepoZero-Py2JS Hard. The CC-Workflow token total under Qwen3.7-Plus is approximate. "–" denotes an unavailable agent range.

Harness	Harness type	Total tokens	Tokens per task	Average agents per task	Agent range
GPT-5.6 Luna High (OpenAI, 2026)					
OpenHands-bash	Single agent	8.98M	0.07M	1.00	1–1
CrewAI	Sequential	60.15M	0.43M	3.00	–
CC-Workflow	Central planning	682.78M	4.91M	5.05	1–11
Omnigent	Node recursive	266.86M	1.92M	4.50	1–8
NanoMA	**Node autonomous**	**490.69M**	**3.53M**	**6.82**	**4–79**
Claude Sonnet 5 (Anthropic, 2026)					
OpenHands-bash	Single agent	34.45M	0.25M	1.00	1–1
CrewAI	Sequential	25.33M	0.18M	3.00	–
CC-Workflow	Central planning	188.70M	1.36M	5.09	2–10
Omnigent	Node recursive	465.91M	3.35M	4.07	3–7
NanoMA	**Node autonomous**	**514.07M**	**3.70M**	**6.80**	**4–23**
Qwen3.7-Plus (Alibaba Cloud, 2026)					
OpenHands-bash	Single agent	15.29M	0.11M	1.00	1–1
CrewAI	Sequential	176.67M	1.27M	4.22	1–12
CC-Workflow	Central planning	177.98M	1.28M	3.40	0–10
Omnigent	Node recursive	551.00M	3.96M	3.94	1–5
NanoMA	**Node autonomous**	**287.72M**	**2.07M**	**3.94**	**1–10**

With GPT-5.6 Luna High, NanoMA outperforms CC-Workflow using fewer tokens, but uses more resources than other baselines. With Claude Sonnet 5, its leading scores accompany the highest token use and average agent count. With Qwen3.7-Plus, it uses fewer tokens than Omnigent at the same average agent count, with higher Micro but lower all-pass. Resource use alone therefore does not predict performance.

Execution time across backbones. In the paired tasks of Table 8, Claude Sonnet 5 reduces mean time for the three baselines but increases it for NanoMA; task-wise fractions show that changes are not uniform.

Table 8: Paired execution times on RepoZero-Py2JS Hard. Means are in minutes and use the same tasks for both backbones within each row. Change is relative to GPT-5.6 Luna; negative values indicate lower mean time with Claude Sonnet 5. The final column reports the fraction of paired tasks on which Claude Sonnet 5 is faster. Paired task sets differ across harnesses.

Harness	Paired tasks	GPT-5.6 Luna mean time	Claude Sonnet 5 mean time	Time change	Claude Sonnet 5 faster
CrewAI	139	9.81	3.42	-65.1%	139/139 (100.0%)
CC-Workflow	134	33.73	11.22	-66.7%	121/134 (90.3%)
Omnigent	129	9.23	8.41	-8.9%	88/129 (68.2%)
NanoMA	**139**	**9.17**	**11.10**	$\mathbf{+21.0\%}$	**54/139 (38.8%)**

Elapsed time also depends on response speed, parallelism, and synchronization; the aggregates do not isolate their contributions. Because task sets differ across harnesses and correctness is not matched, these data support within-harness backbone comparisons, not a controlled cross-harness speed ranking.

D IMPLICATIONS AND LIMITATIONS

Information value rather than agent count. NanoMA's collaborative value lies in improving the information available for agent decisions, not merely increasing agent numbers. Useful branches should produce distinct, comparable evidence. `query` retrieves that evidence on demand, while `send` delivers findings when they can still affect another agent's decisions. The communication ablations in Sections 4.3.1 and 4.3.2 support broad information access and selective message delivery in the evaluated settings; they do not imply that more communication is always better. Relevance and timing matter alongside information availability.

Local decisions with wider consequences. Giving each agent topology actions does not ensure that it uses them well. A new branch can contribute independent evidence but also require additional coordination; waiting can resolve a dependency but sacrifice parallel work; and pruning can save computation but discard evidence that has not yet been shared. These effects extend beyond the acting agent and can emerge only later. Synchronization should therefore follow actual dependencies, and pruning should preserve useful evidence. Static prompt rules express these principles but do not reliably determine the right action in each execution state.

Scope of the evidence. The reported gains are specific to the evaluated backbones, benchmarks, and budgets. Communication policies can also change agent expansion and subsequent execution, so their ablations measure the resulting system behavior rather than an isolated information-transfer effect. Likewise, the case study in Section 5.4 records parallel execution and a runtime advantage without isolating the causal contribution of parallelism. The cost comparisons in Appendix C do not establish a uniform efficiency advantage across backbones.

Learning action selection from feedback. Future work should train action-selection policies using runtime feedback, such as verified task progress, resource use, and whether shared evidence changes a subsequent decision. The objective is to help agents assess when continued execution is preferable to expansion, communication, or termination, and to account for effects on other agents rather than immediate local progress alone. Assigning credit to actions whose consequences appear later remains an open challenge.